Skua: Extending Distributed Tracing Vertically into the Linux Kernel

Harshal Sheth and Andrew Sun

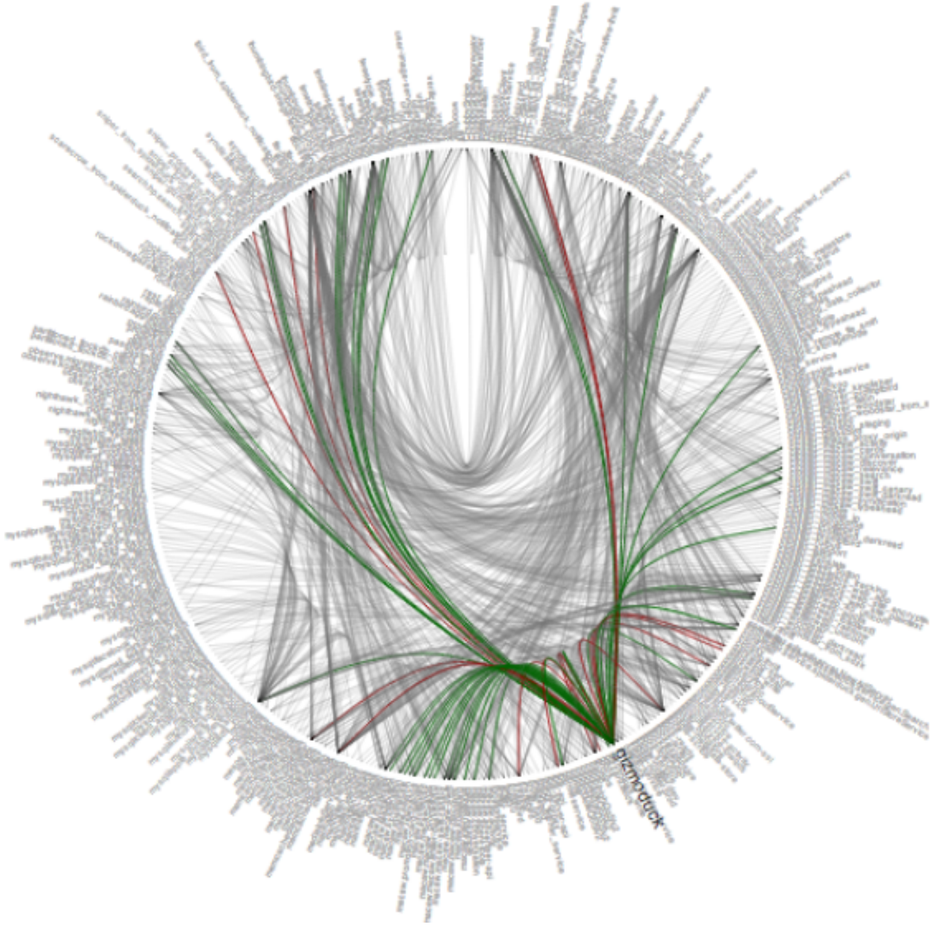
DevConf 2018

Distributed Systems

- Complex applications are no longer monolithic
  - Modular/agile development
  - Continuous deployment
  - Independent scaling

Distributed Systems

- Complex applications are no longer monolithic
  - Modular/agile development
  - Continuous deployment
  - Independent scaling
- Increasingly seen in large companies
- Hard to debug



Twitter, 2013

Example Distributed System

Frontend

Web Results

PageRank

Images

VisualRank

Legend

request

application within a distributed system

request pathway

Example Distributed System

Frontend

Web Results

PageRank

Images

VisualRank

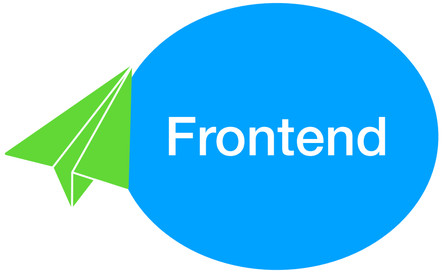
Legend

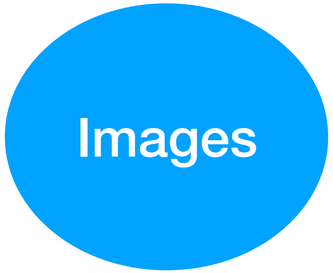
request

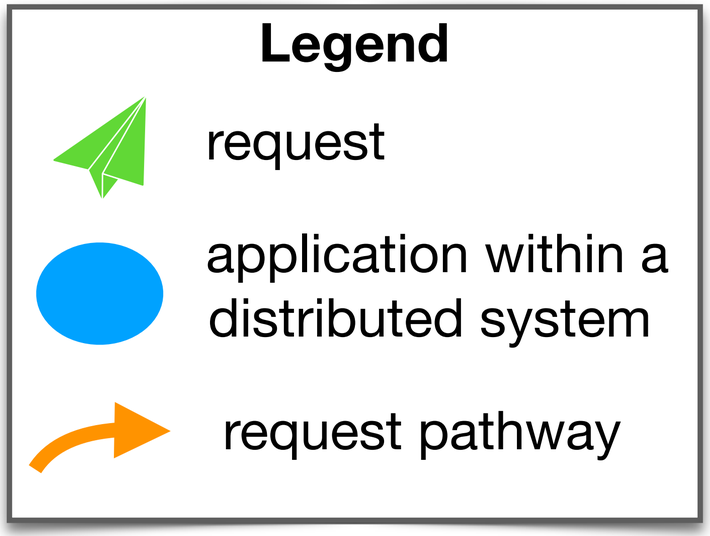
application within a distributed system

request pathway

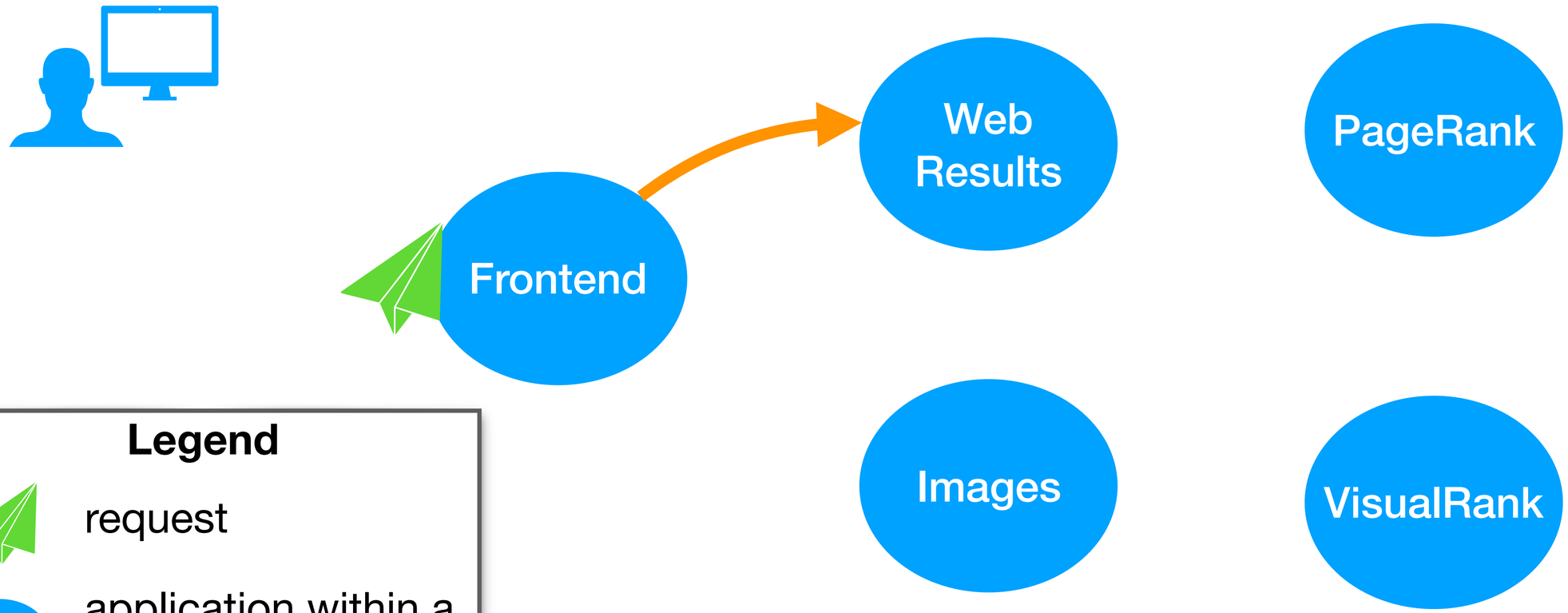
Example Distributed System



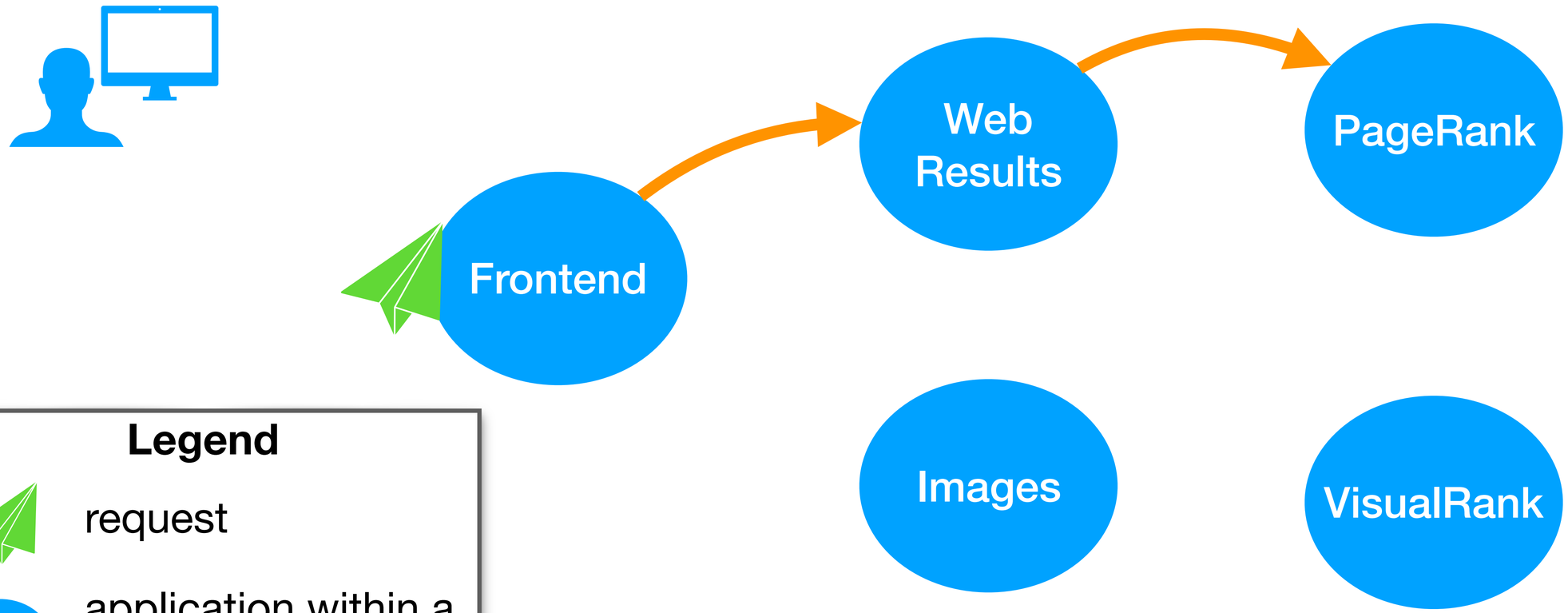




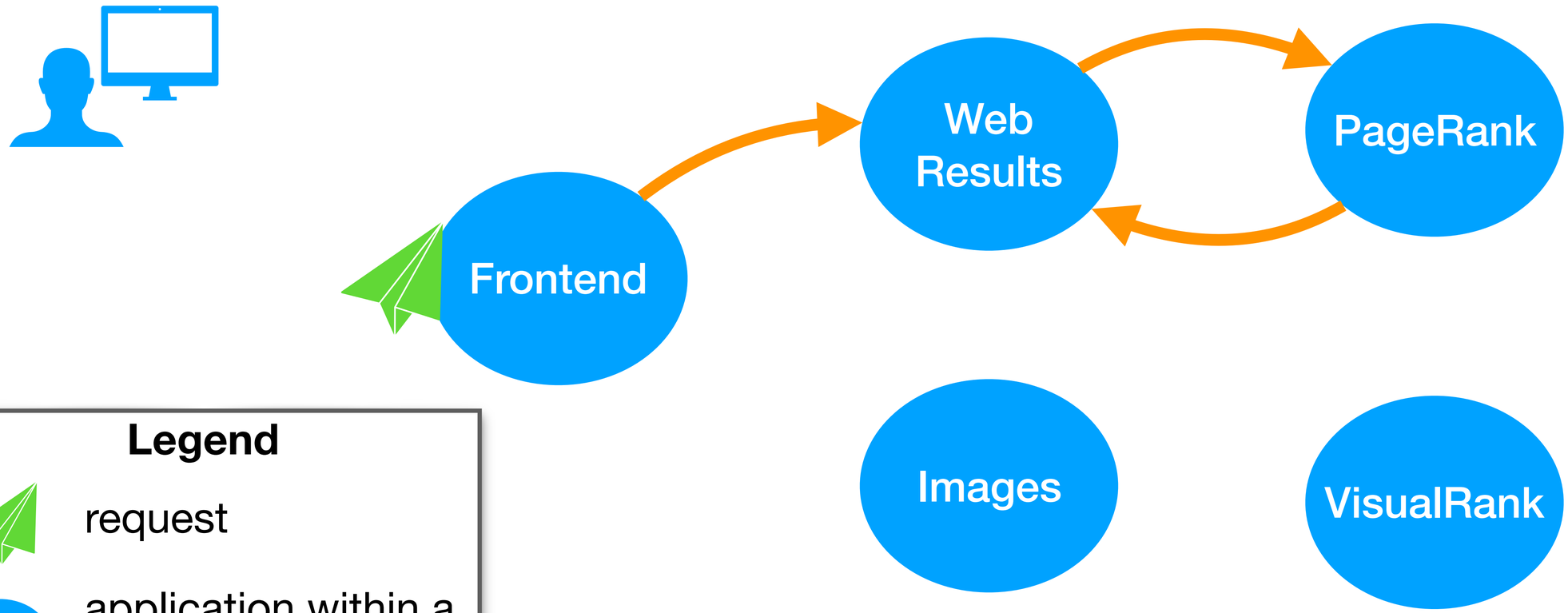
Example Distributed System



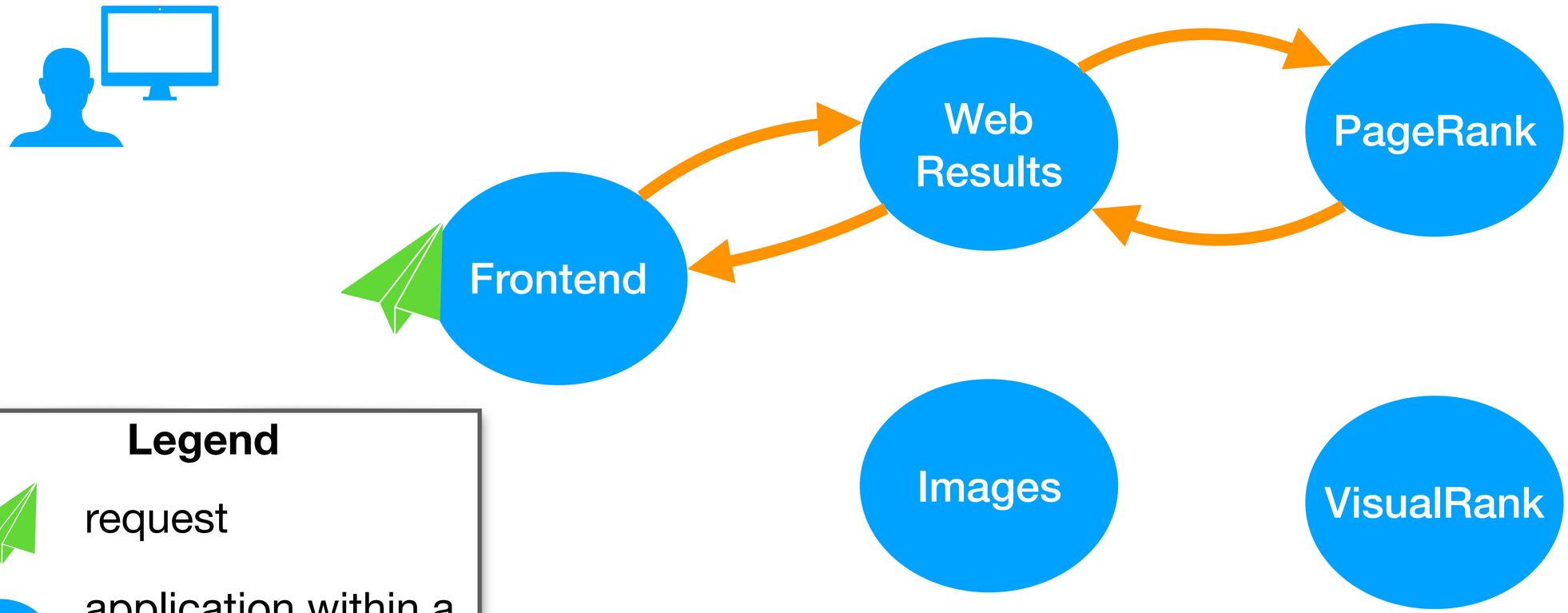
Example Distributed System



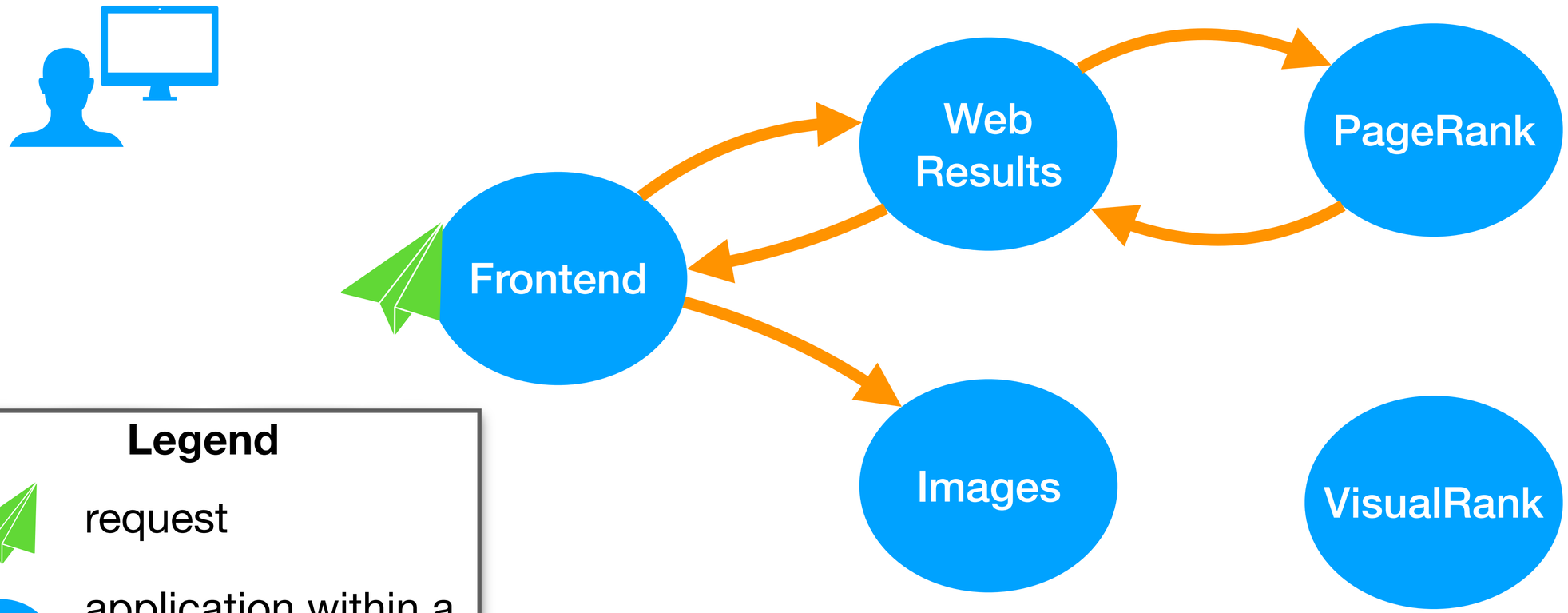
Example Distributed System



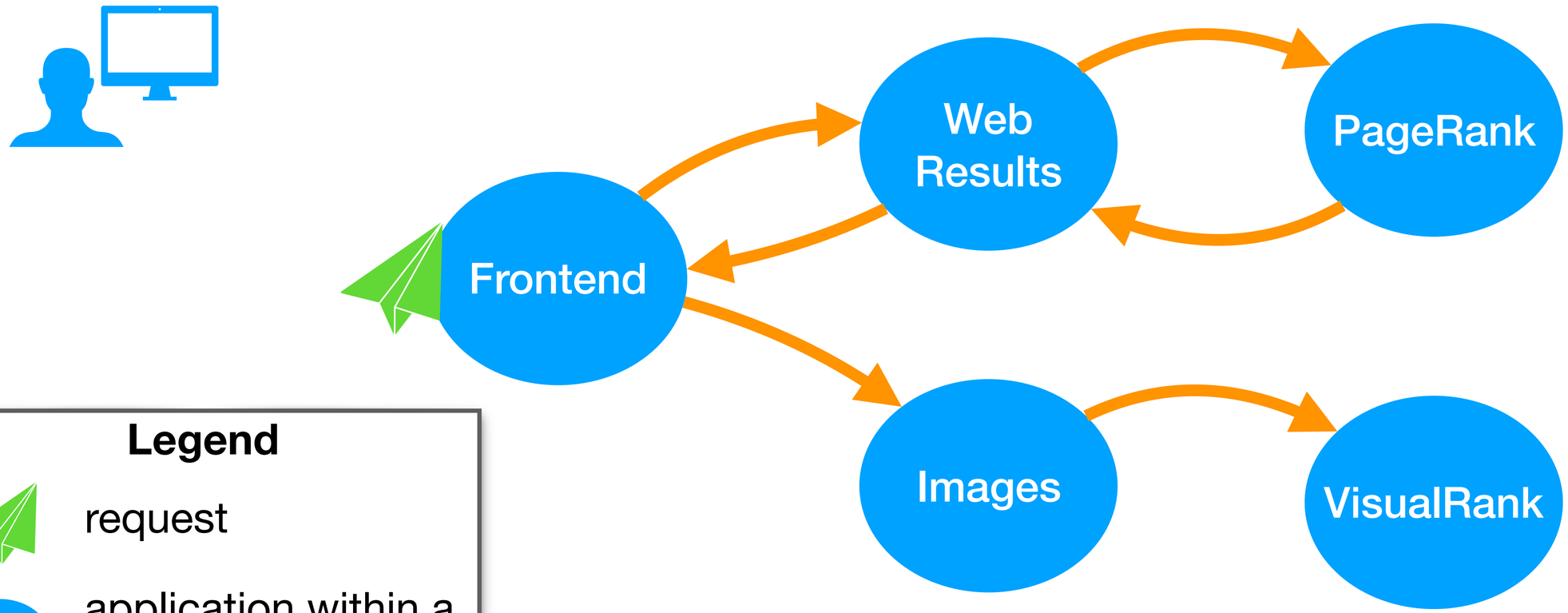
Example Distributed System



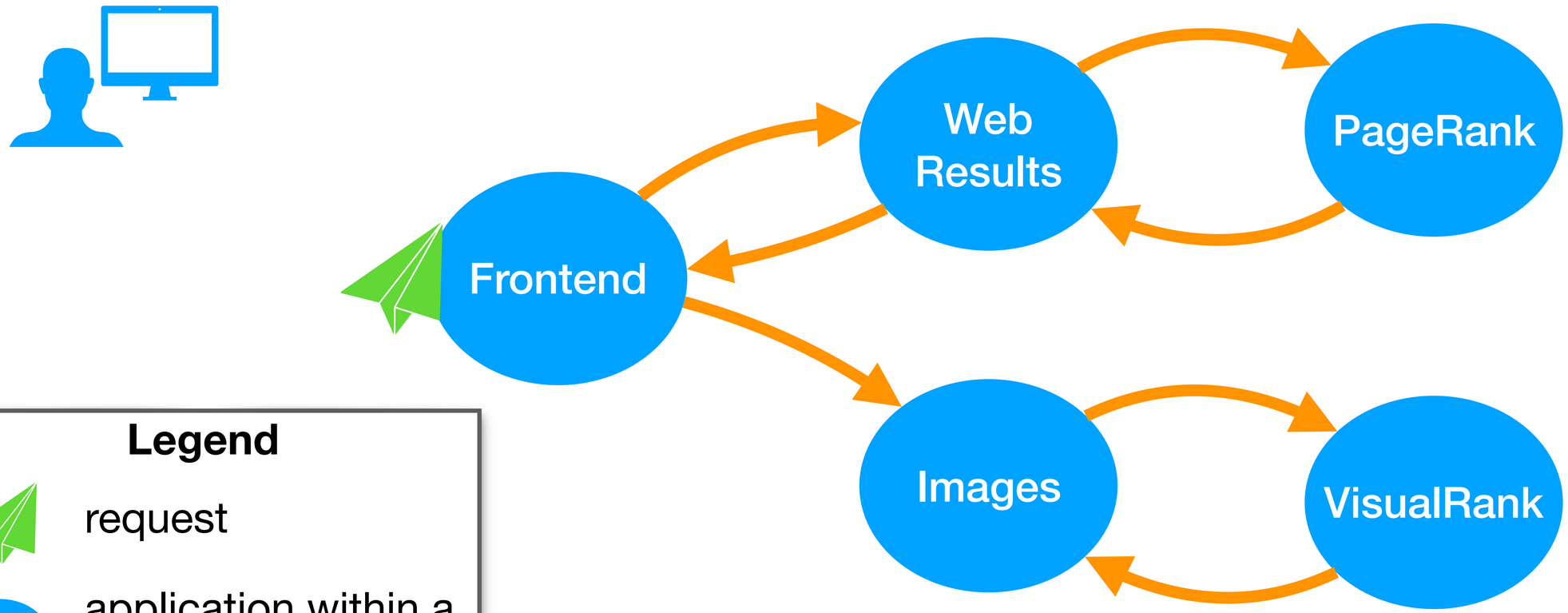
Example Distributed System



Example Distributed System



Example Distributed System



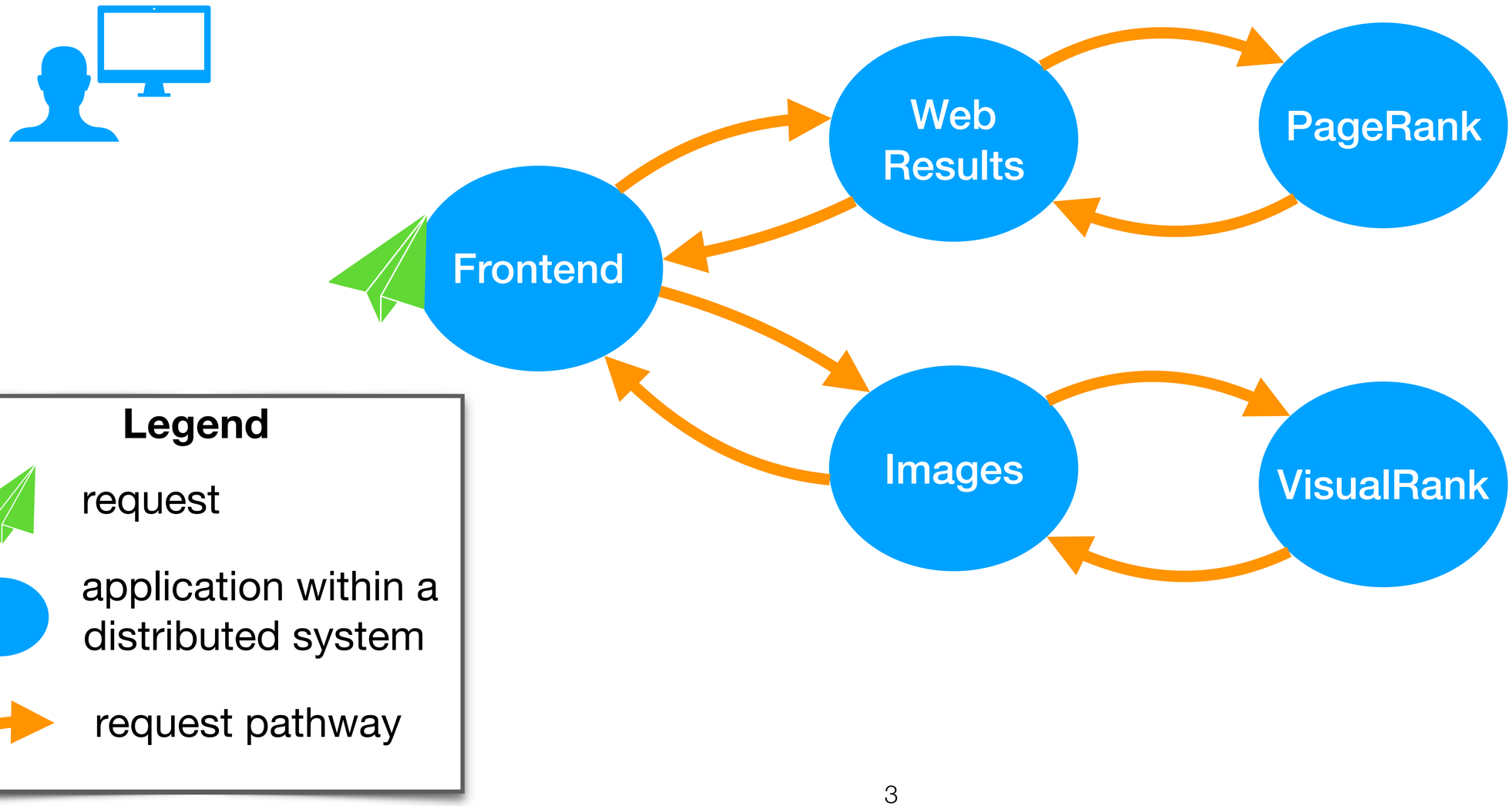
Legend

request

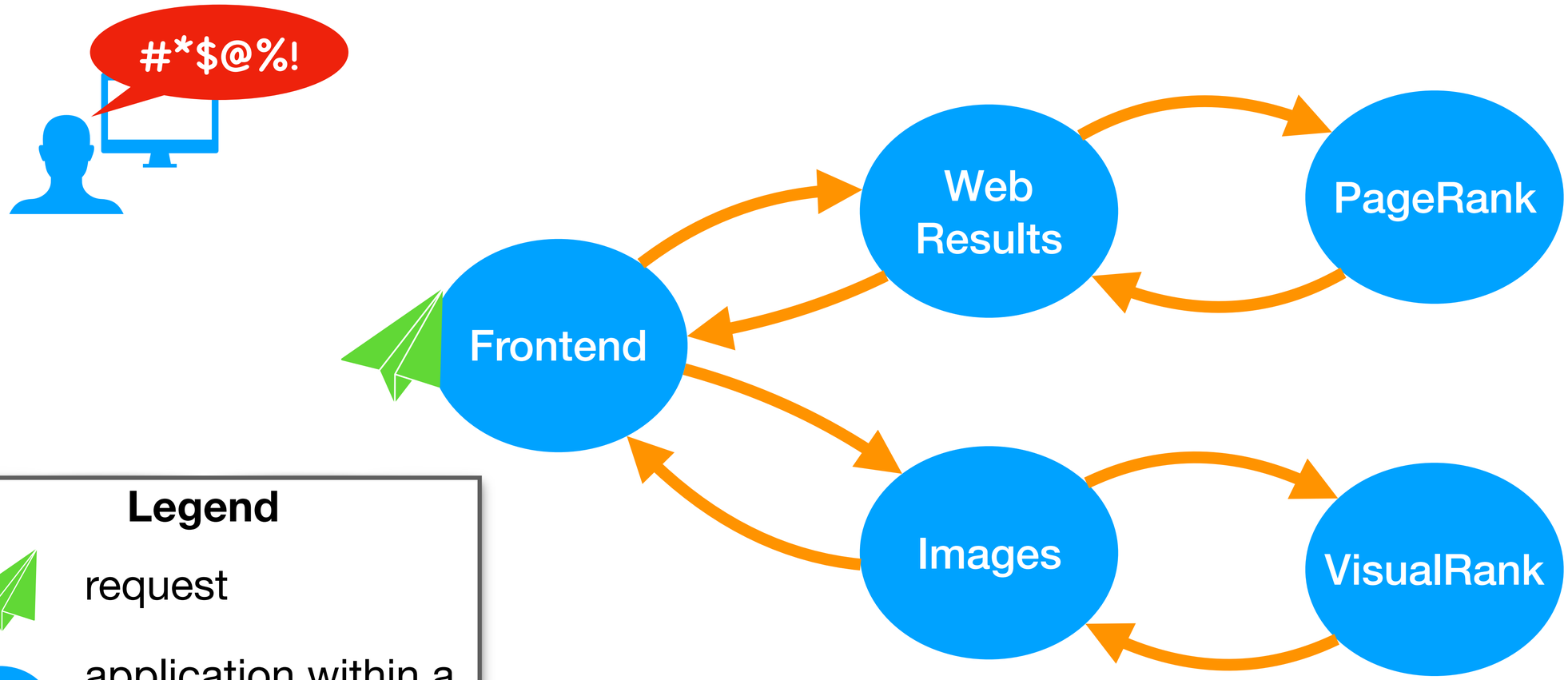
application within a distributed system

request pathway

Example Distributed System



Example Distributed System



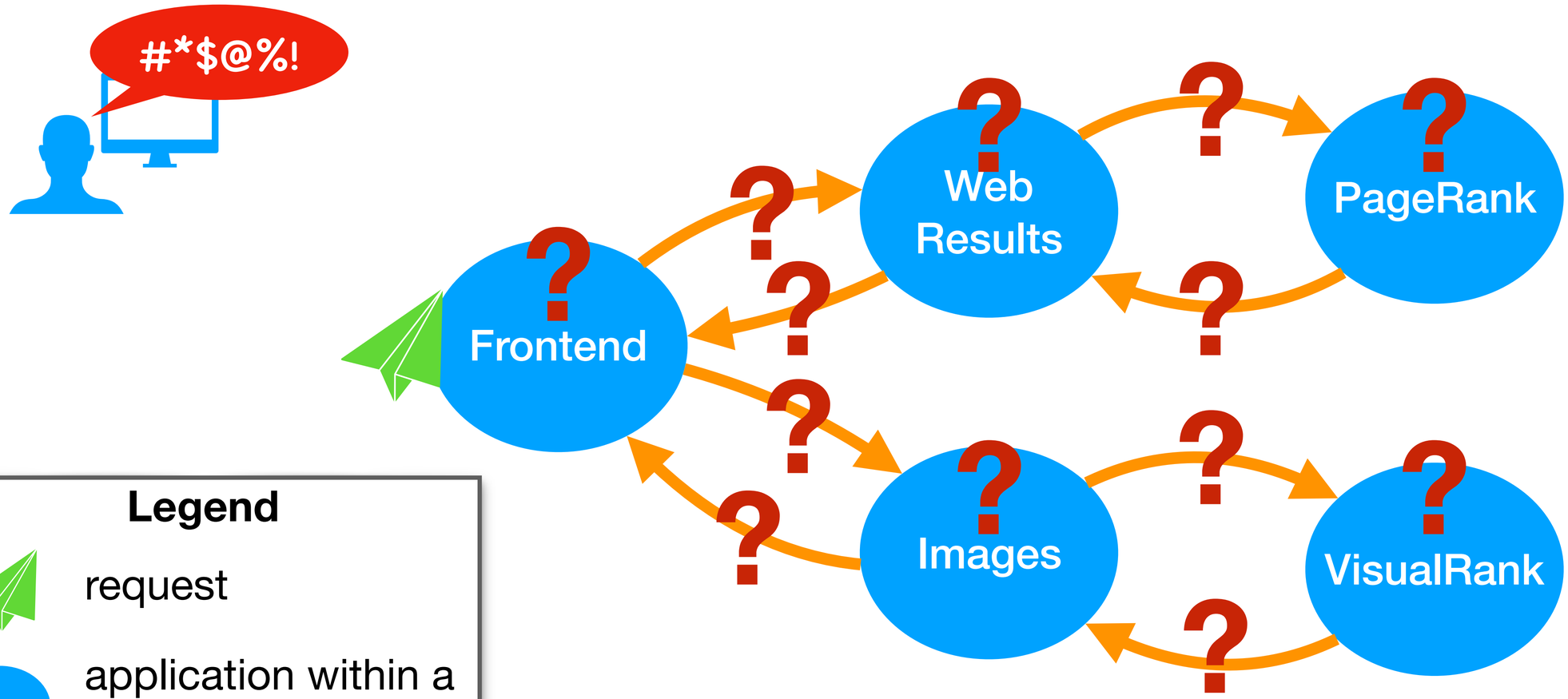
Legend

request

application within a distributed system

request pathway

Example Distributed System



Legend

request

application within a distributed system

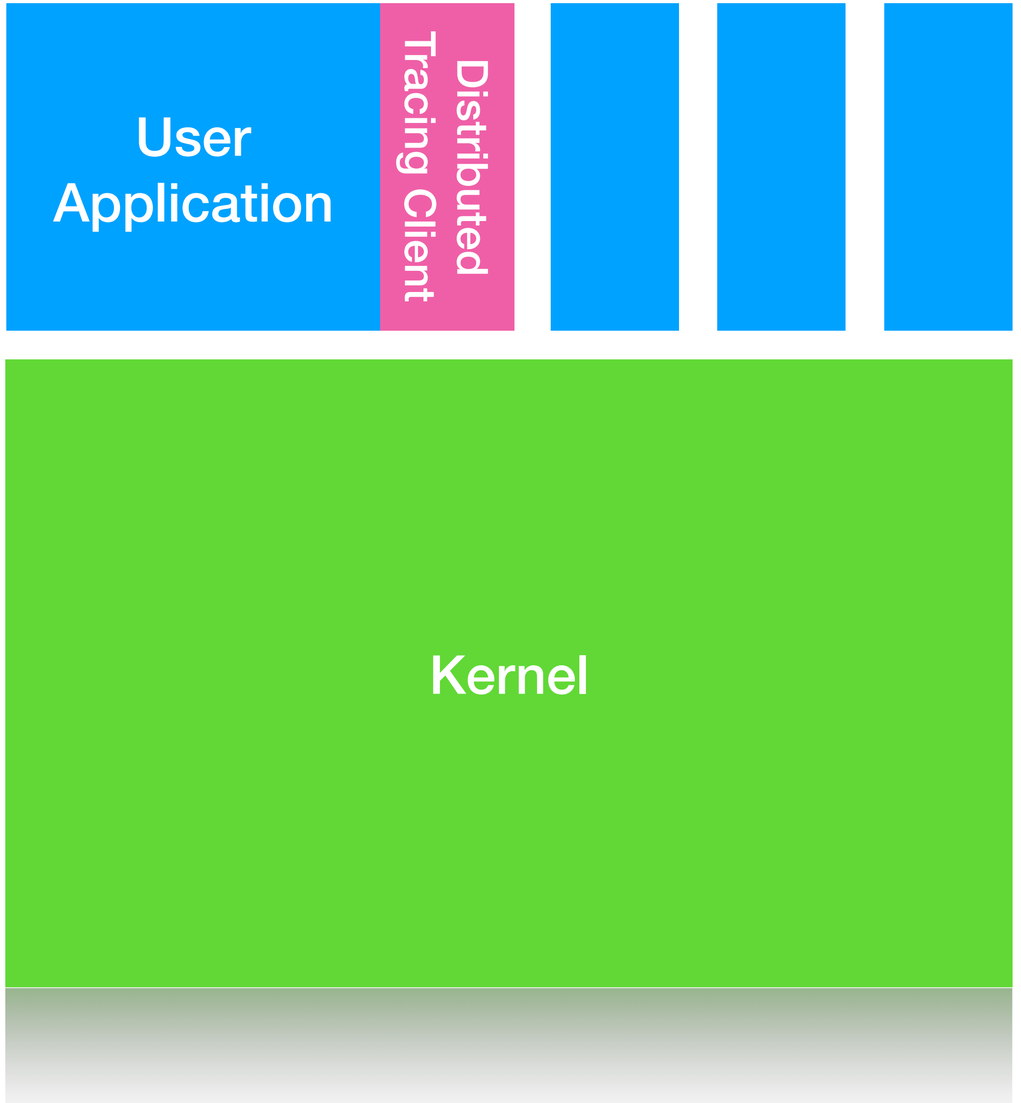
request pathway

Distributed Tracing

- Monitoring and troubleshooting distributed systems
  - Discovering latency issues
  - Graphing service dependencies
  - Root-cause analysis of backend issues
  - Tracing a specific request through the entire system

What does distributed tracing miss?

- There's more to performance than meets the eye of existing distributed tracing tools
  - Contention between applications
  - Kernel bugs
  - Security patches (e.g. Meltdown/ Spectre)
- Can we gain visibility regarding these issues via the kernel?



Our goal: extend distributed tracing into the kernel

Our Approach

Jaeger
distributed tracing
framework from Uber

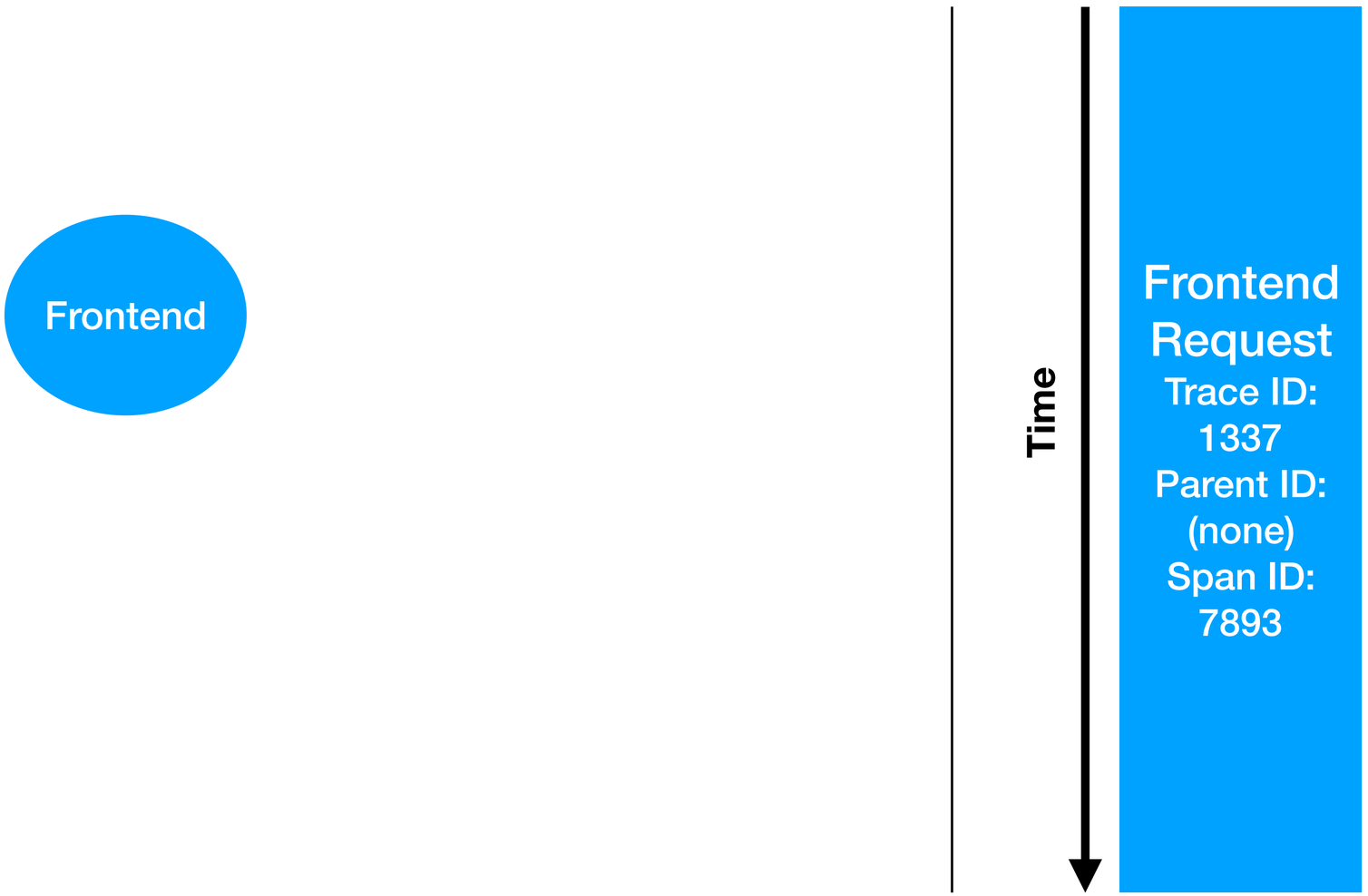
+

LTTng
Linux kernel
trace toolkit

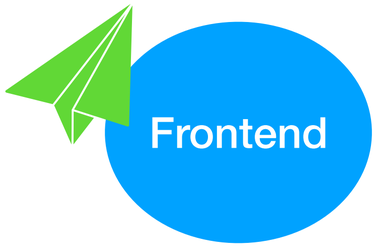
=

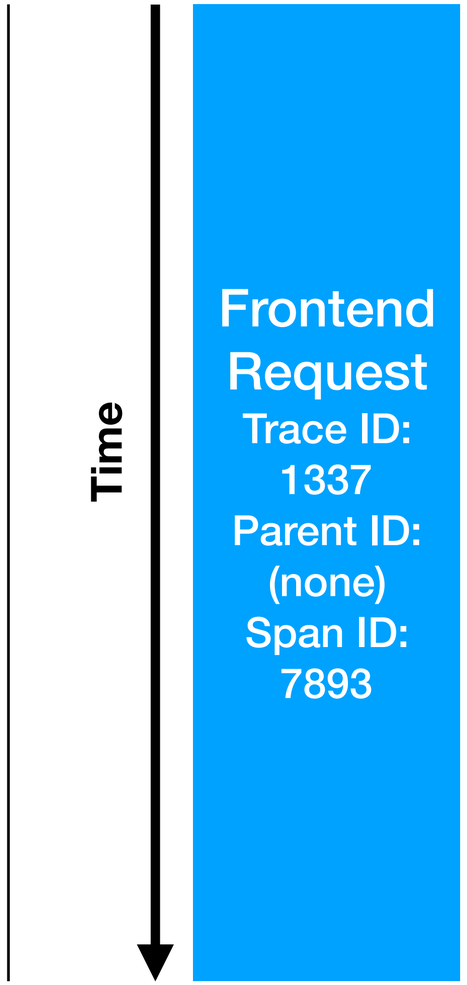
Skua

Traces and Spans

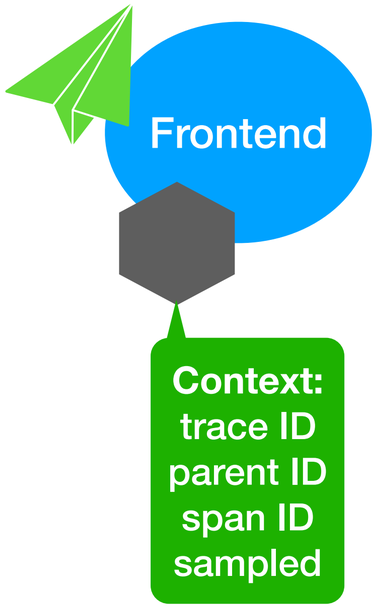


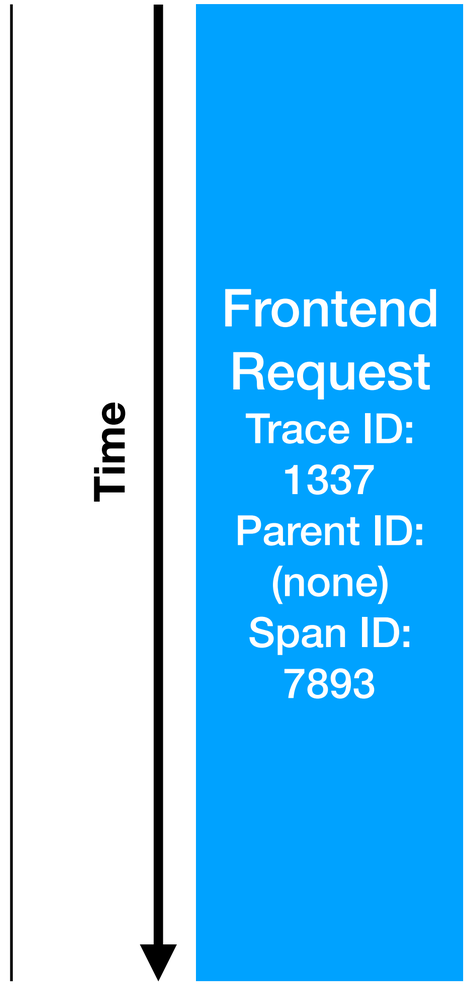
Traces and Spans



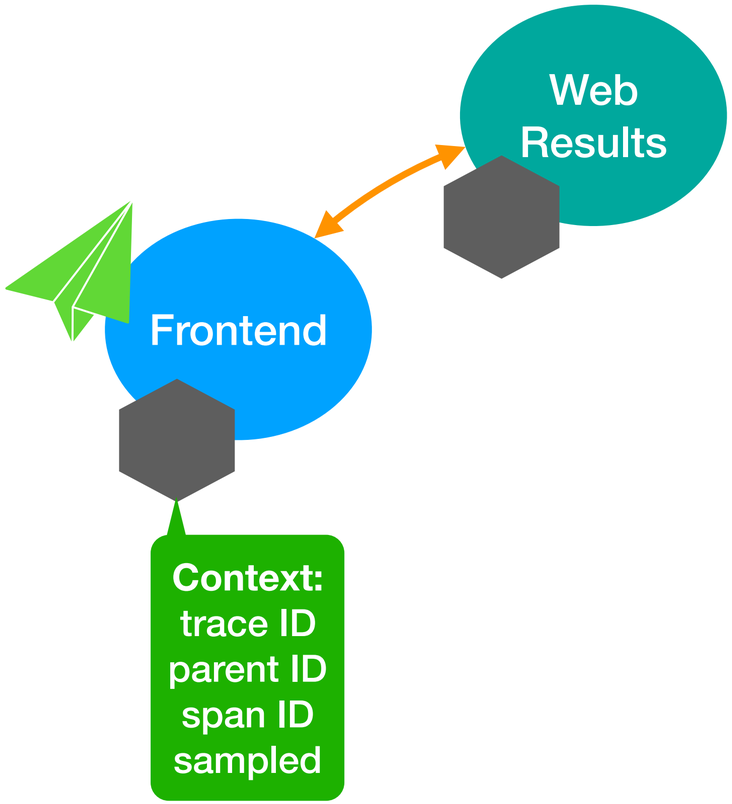


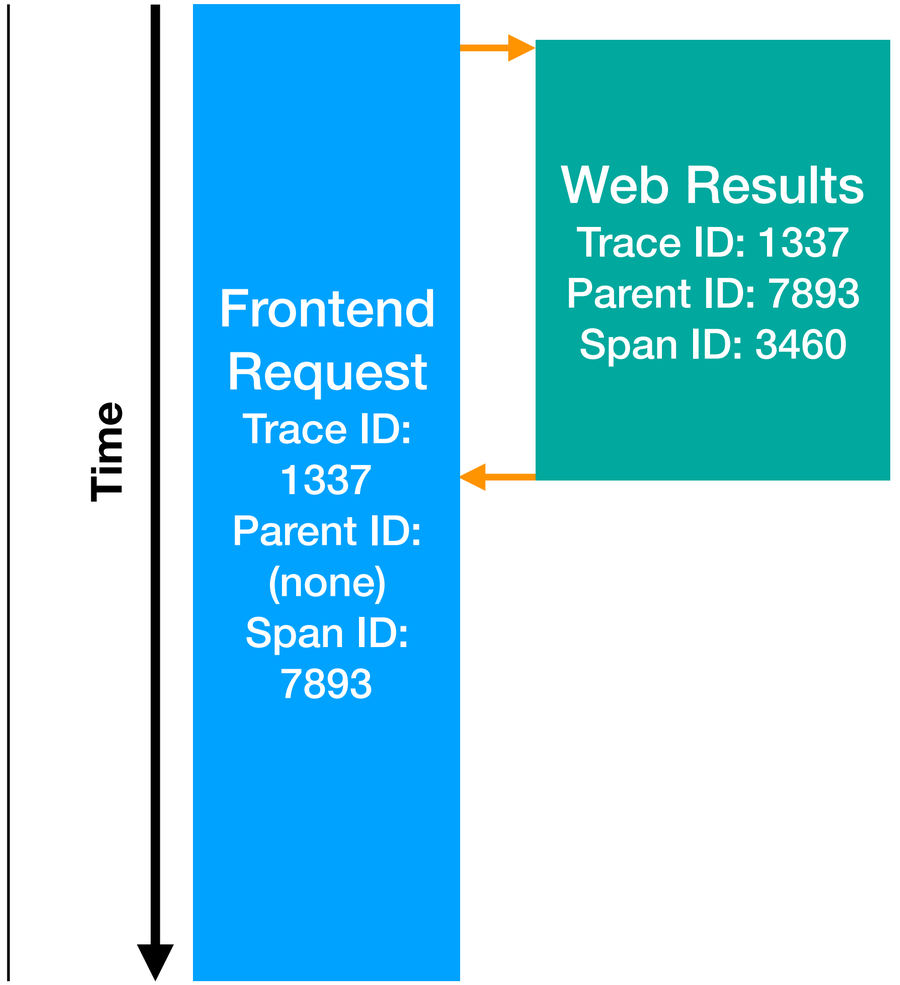
Traces and Spans



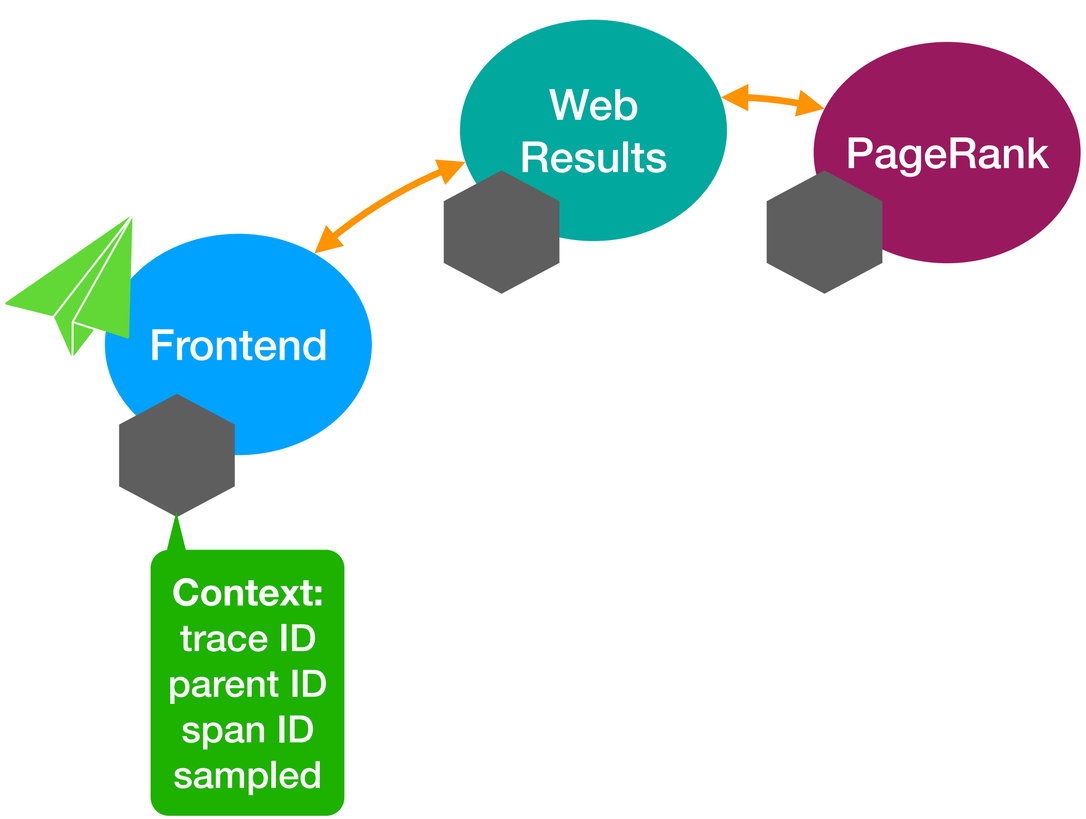


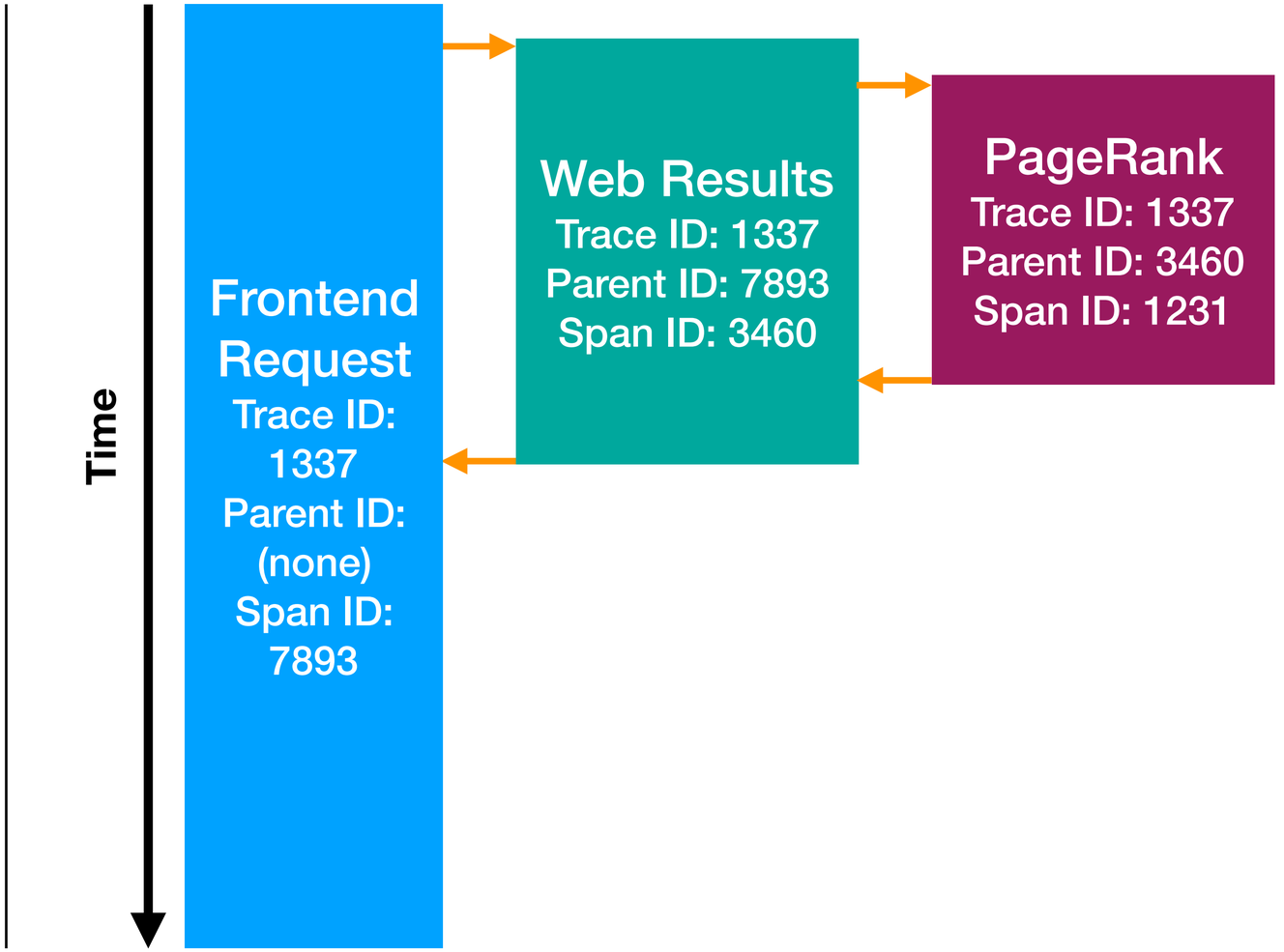
Traces and Spans



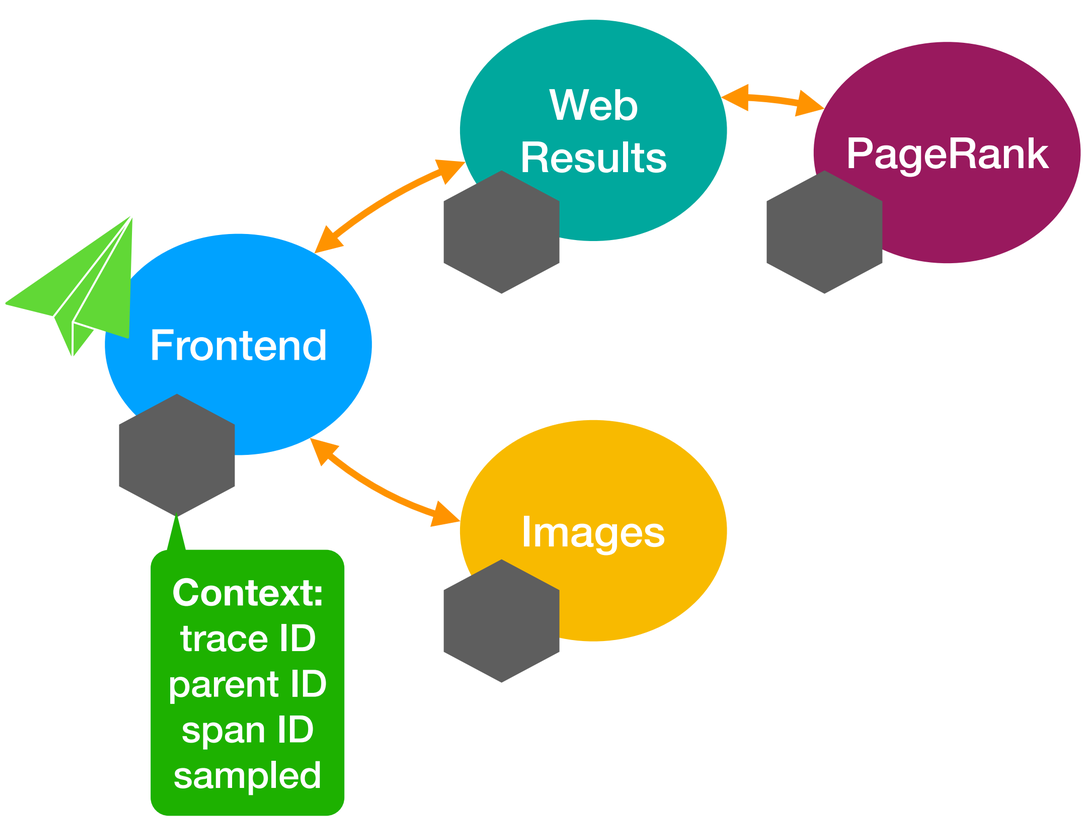


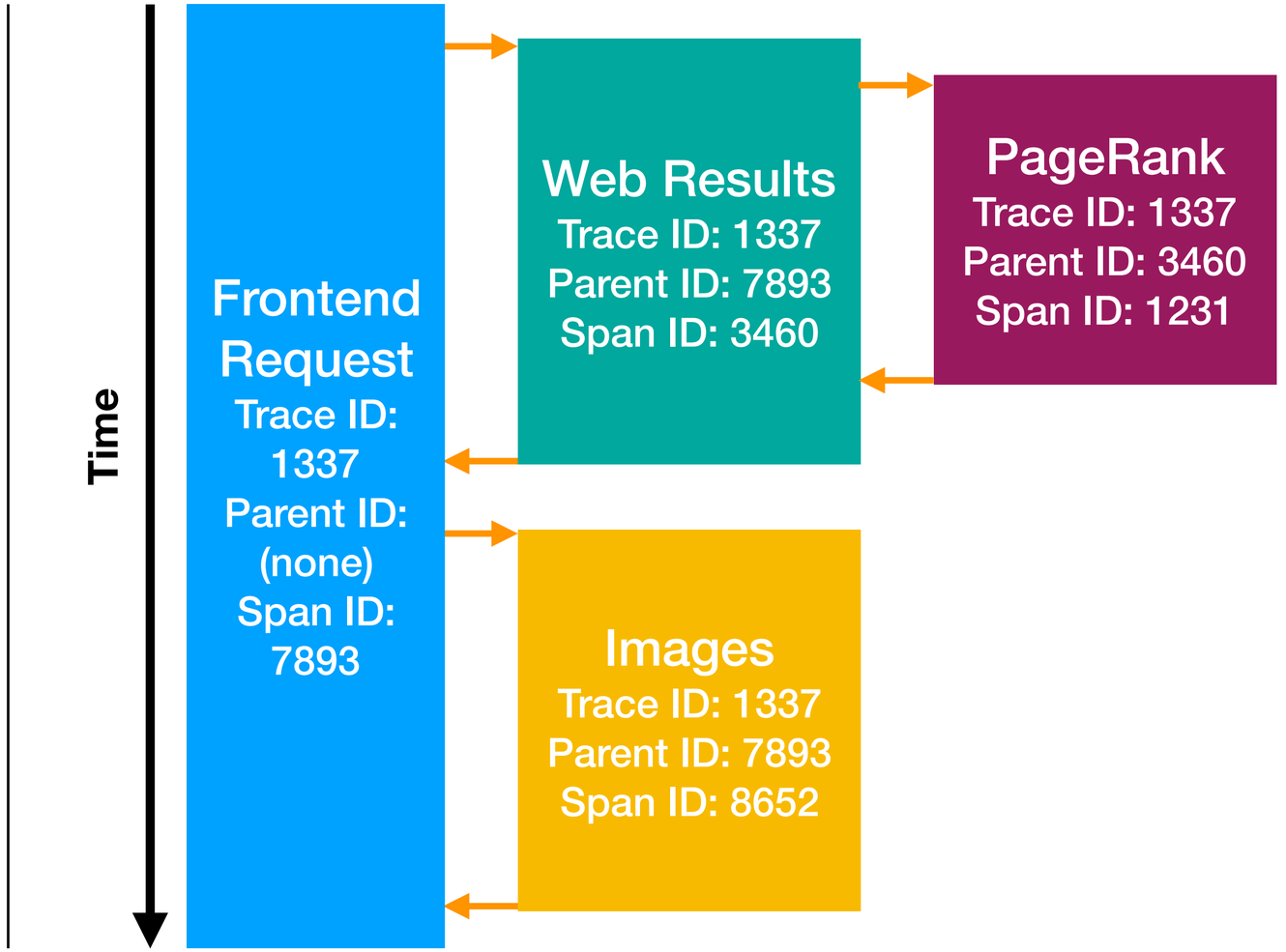
Traces and Spans



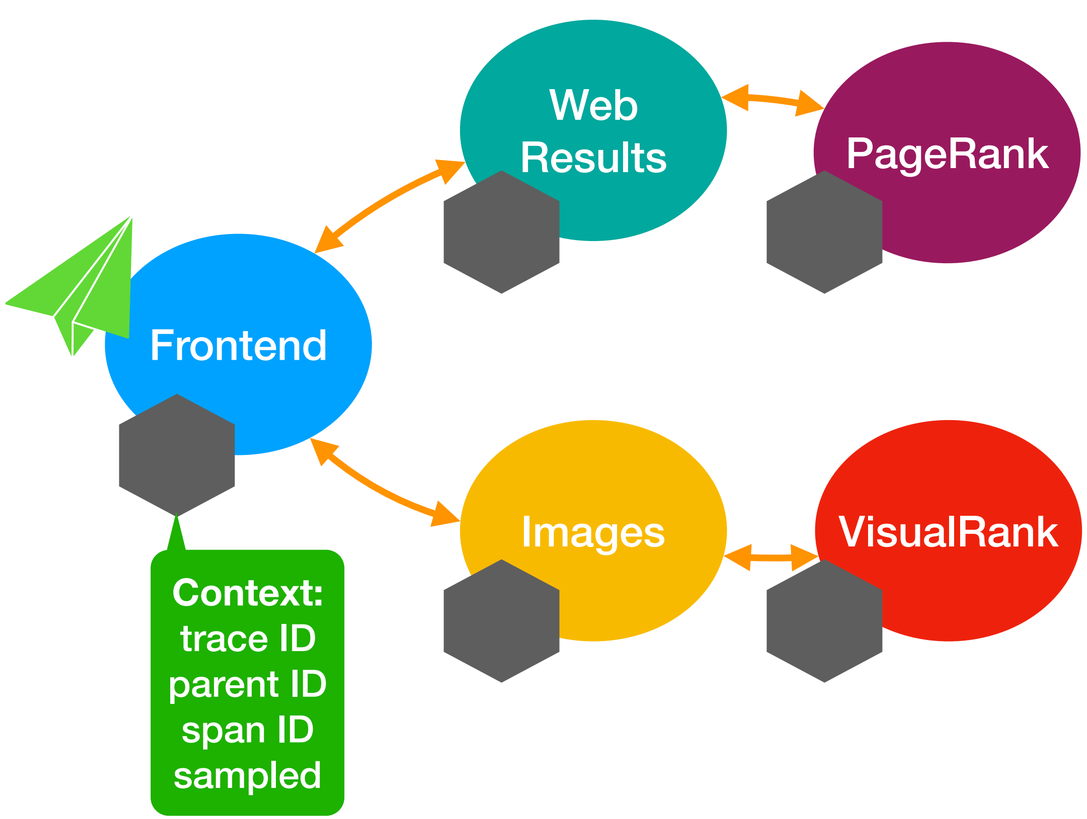


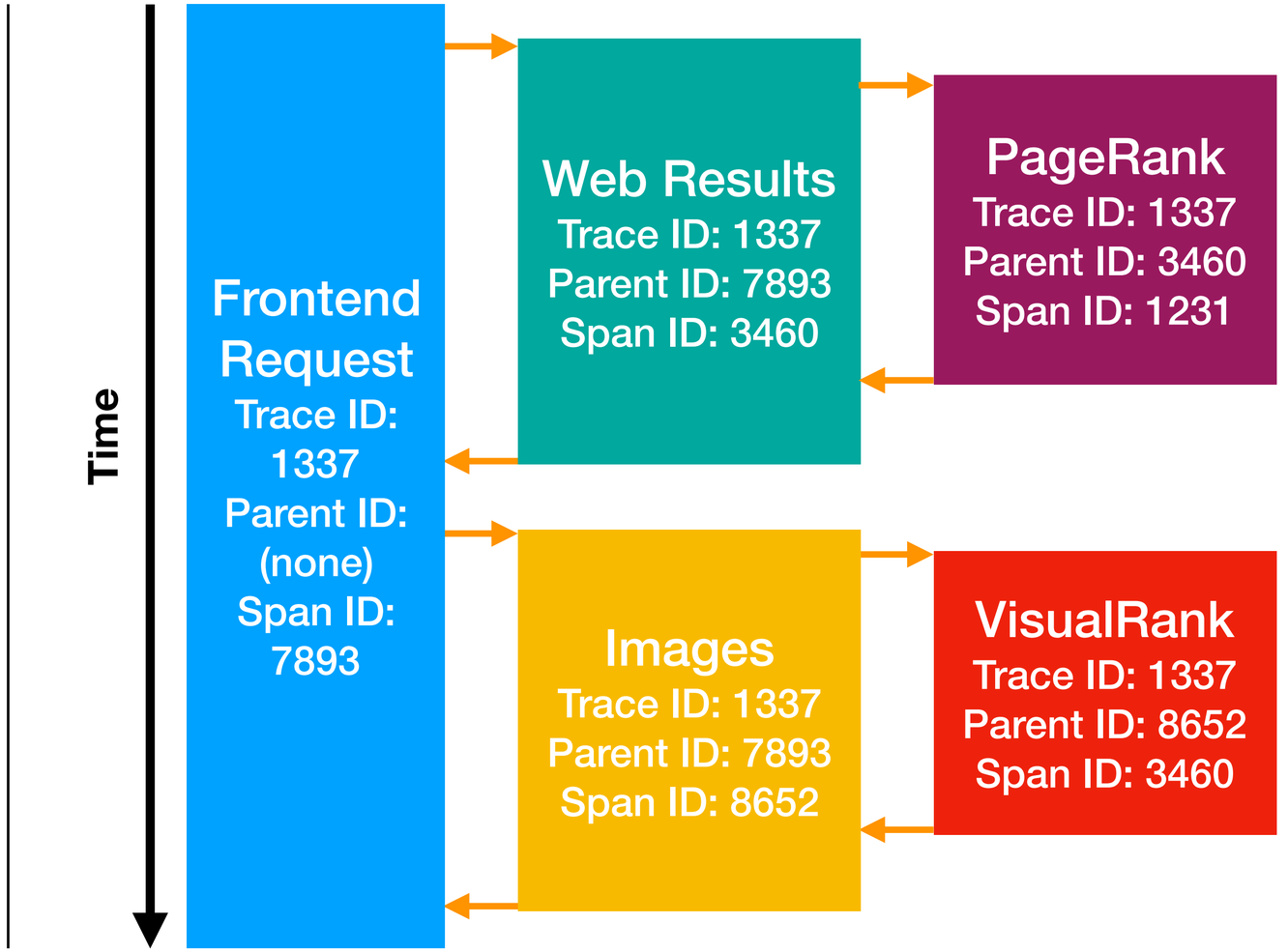
Traces and Spans



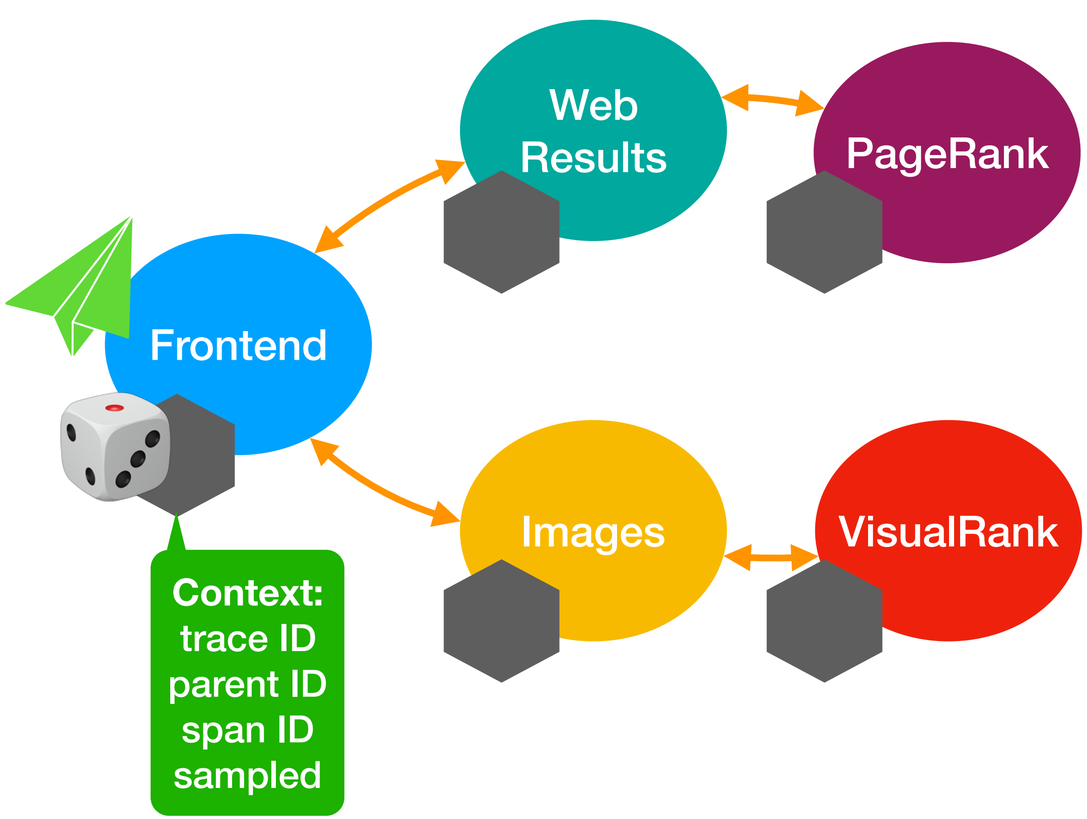


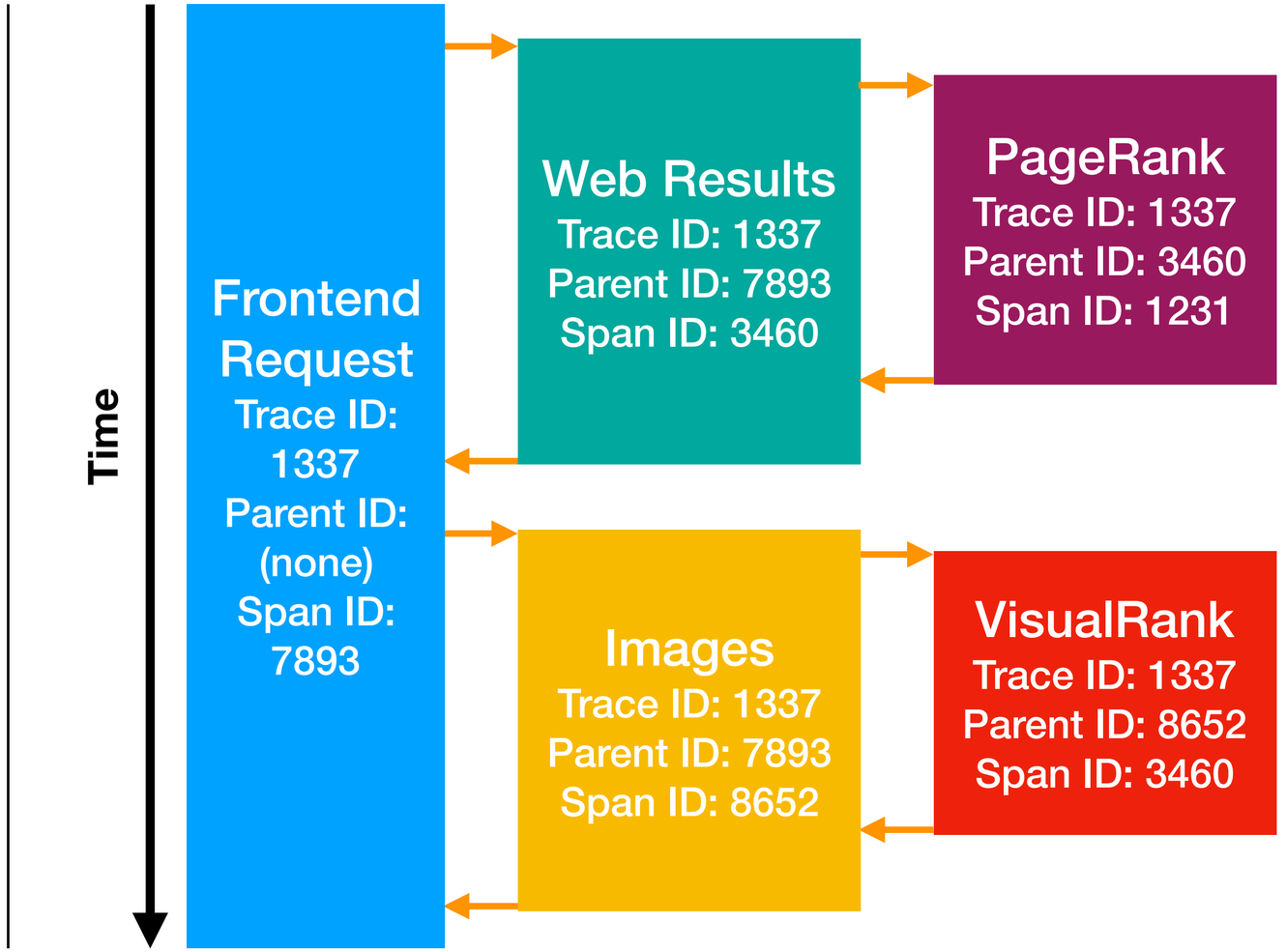
Traces and Spans



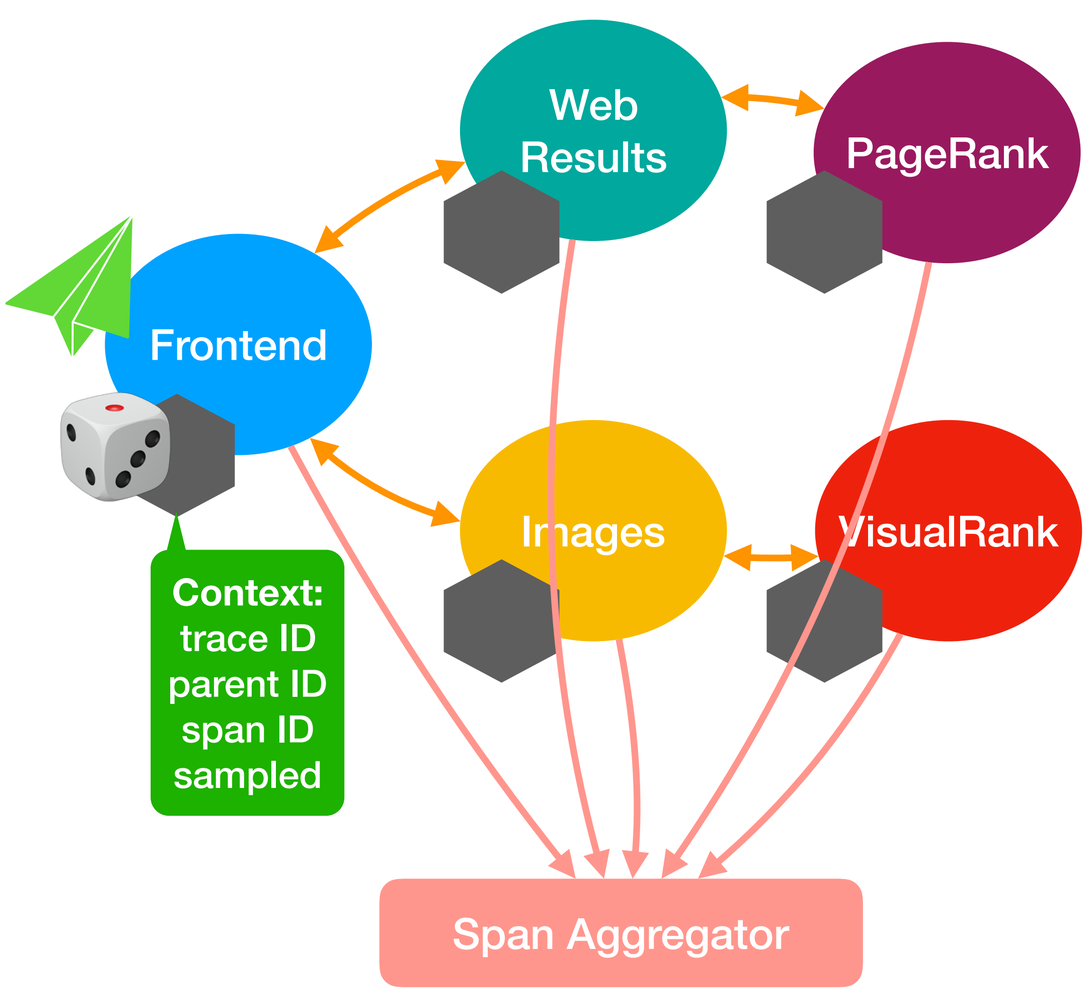


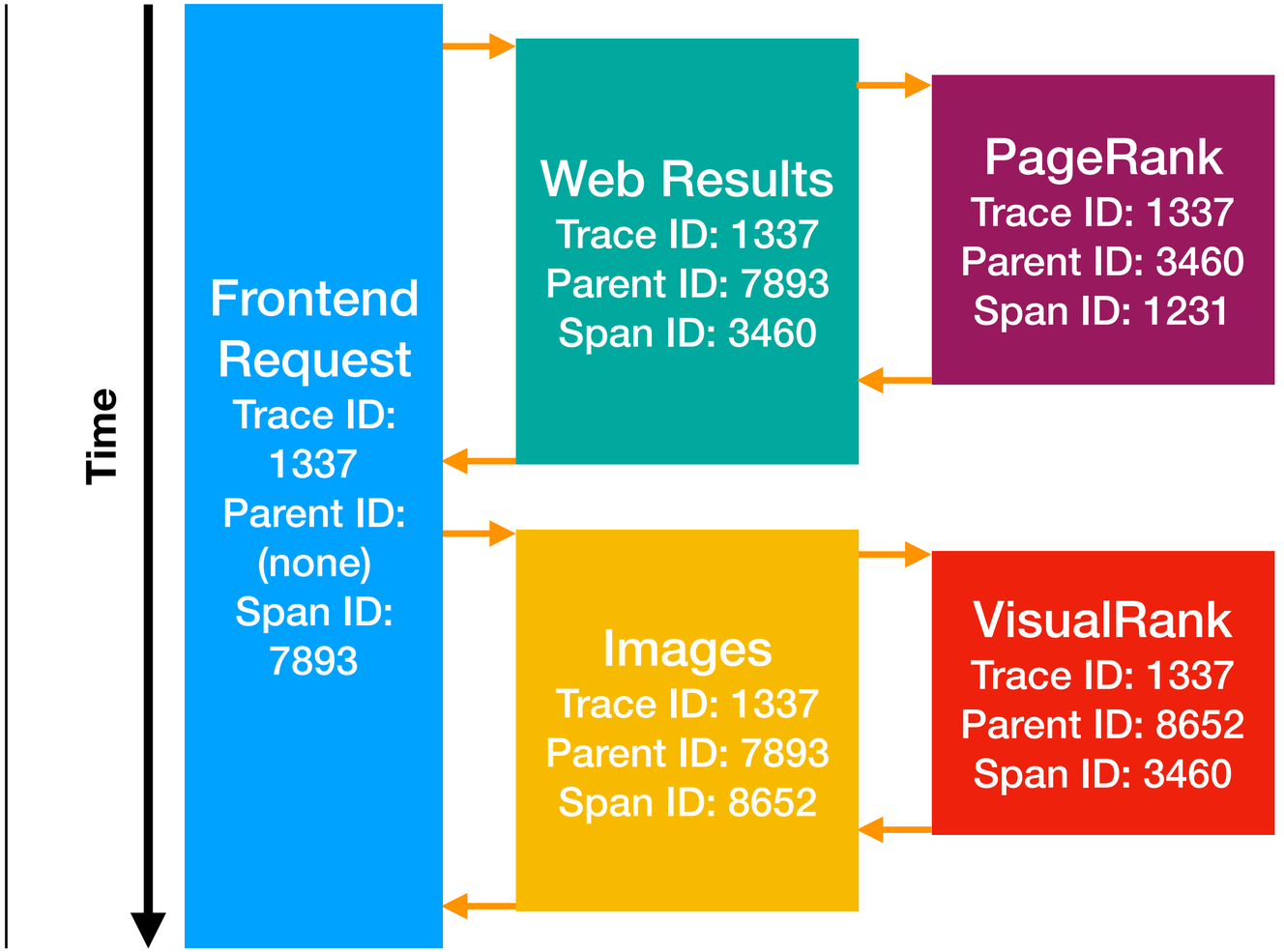
Traces and Spans



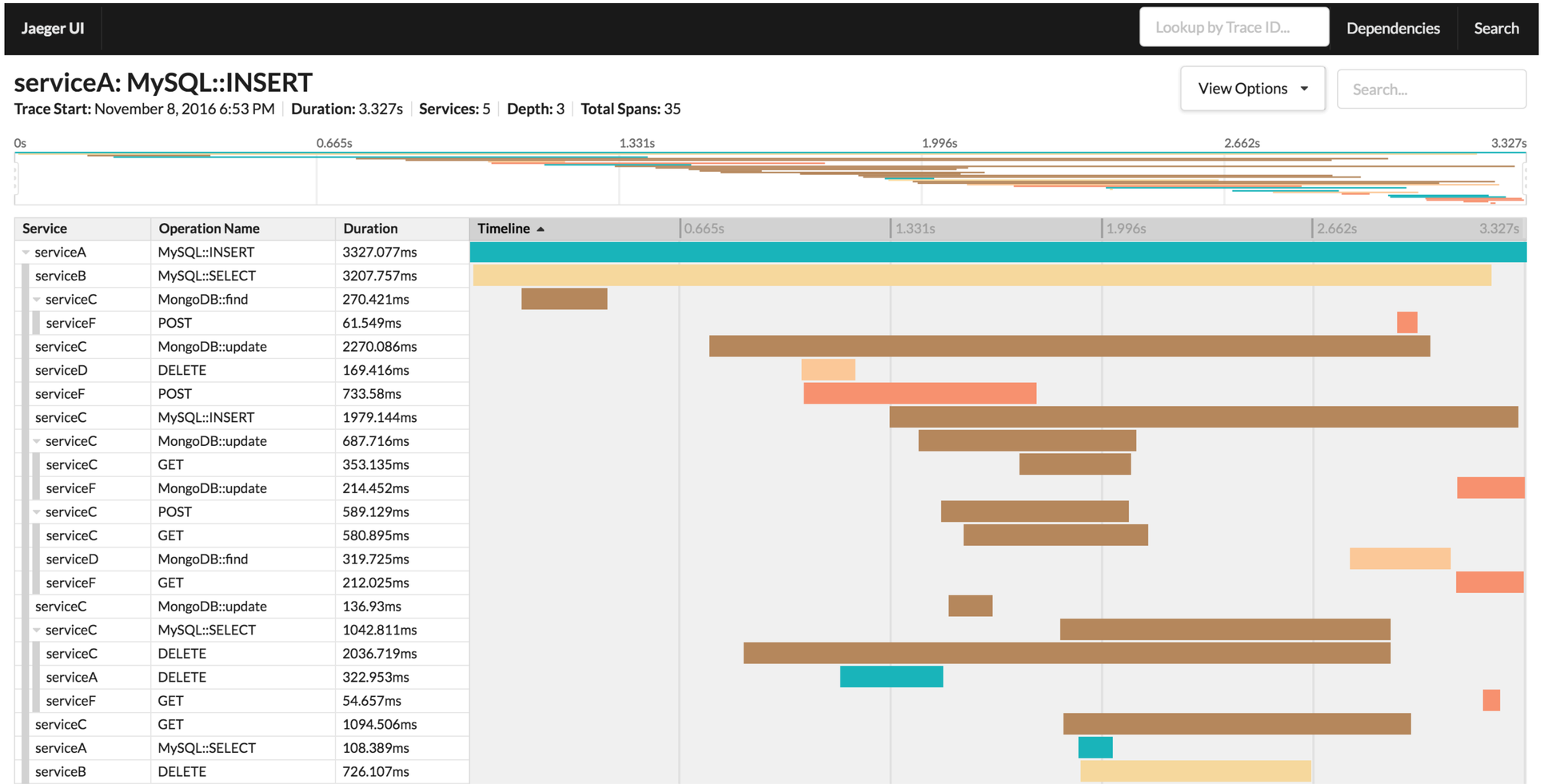


Traces and Spans



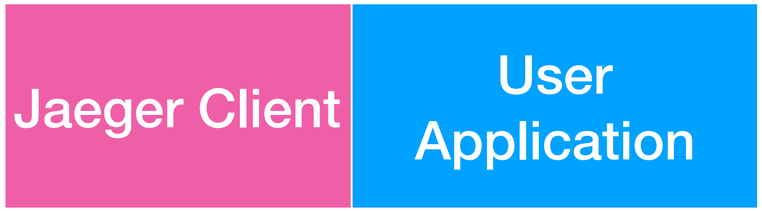


Jaeger

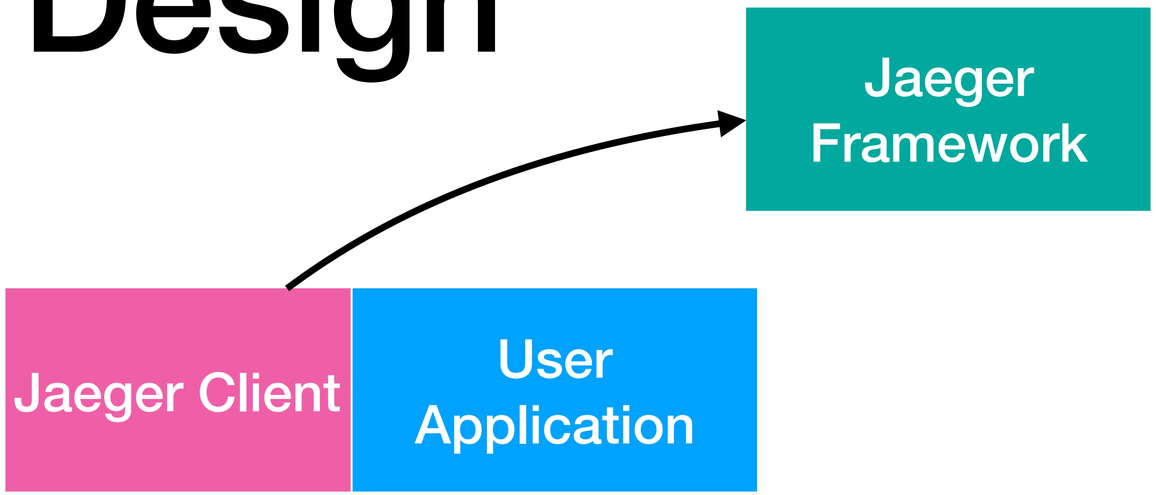


https://eng.uber.com/distributed-tracing/

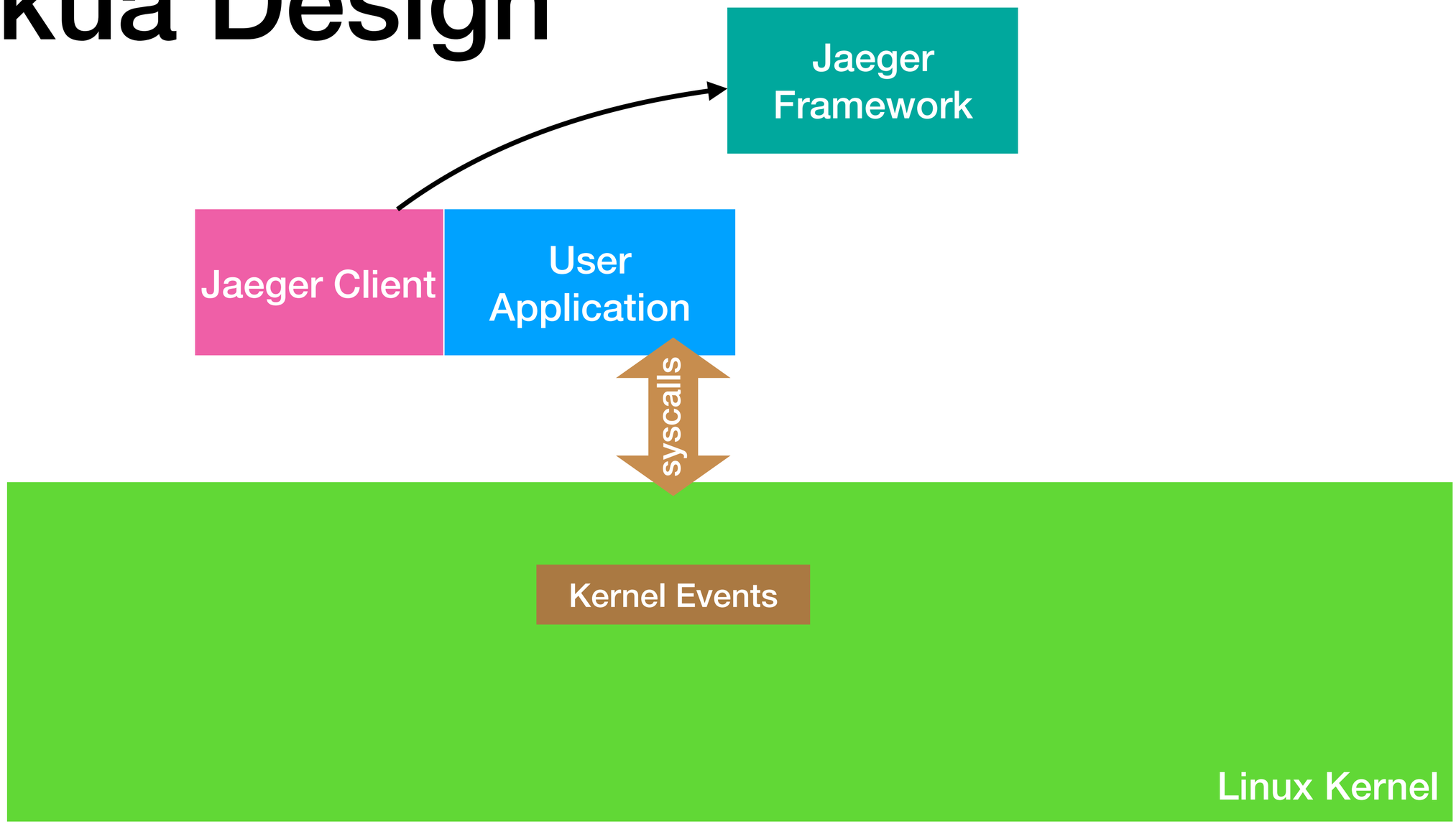
Skua Design



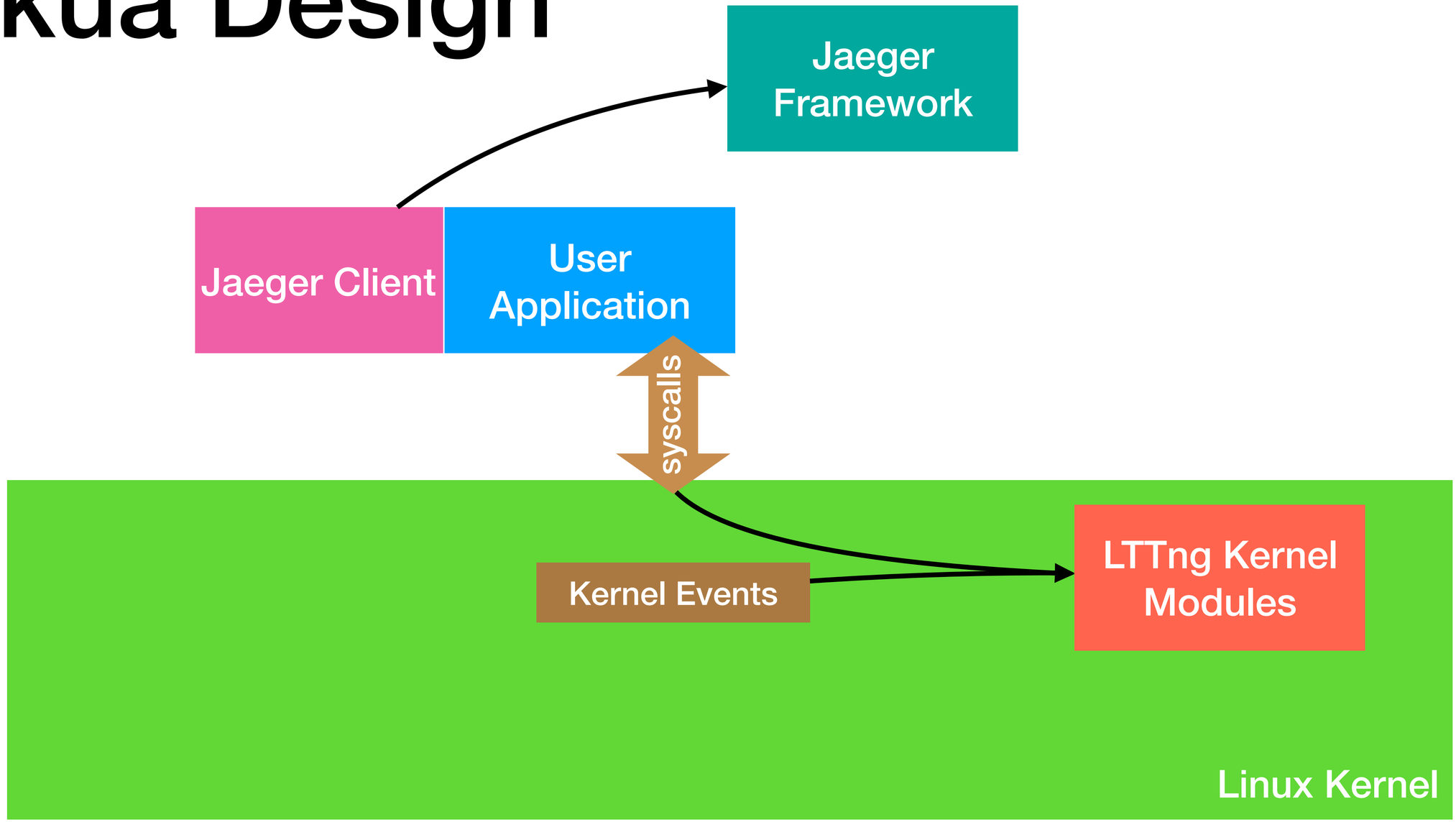
Skua Design



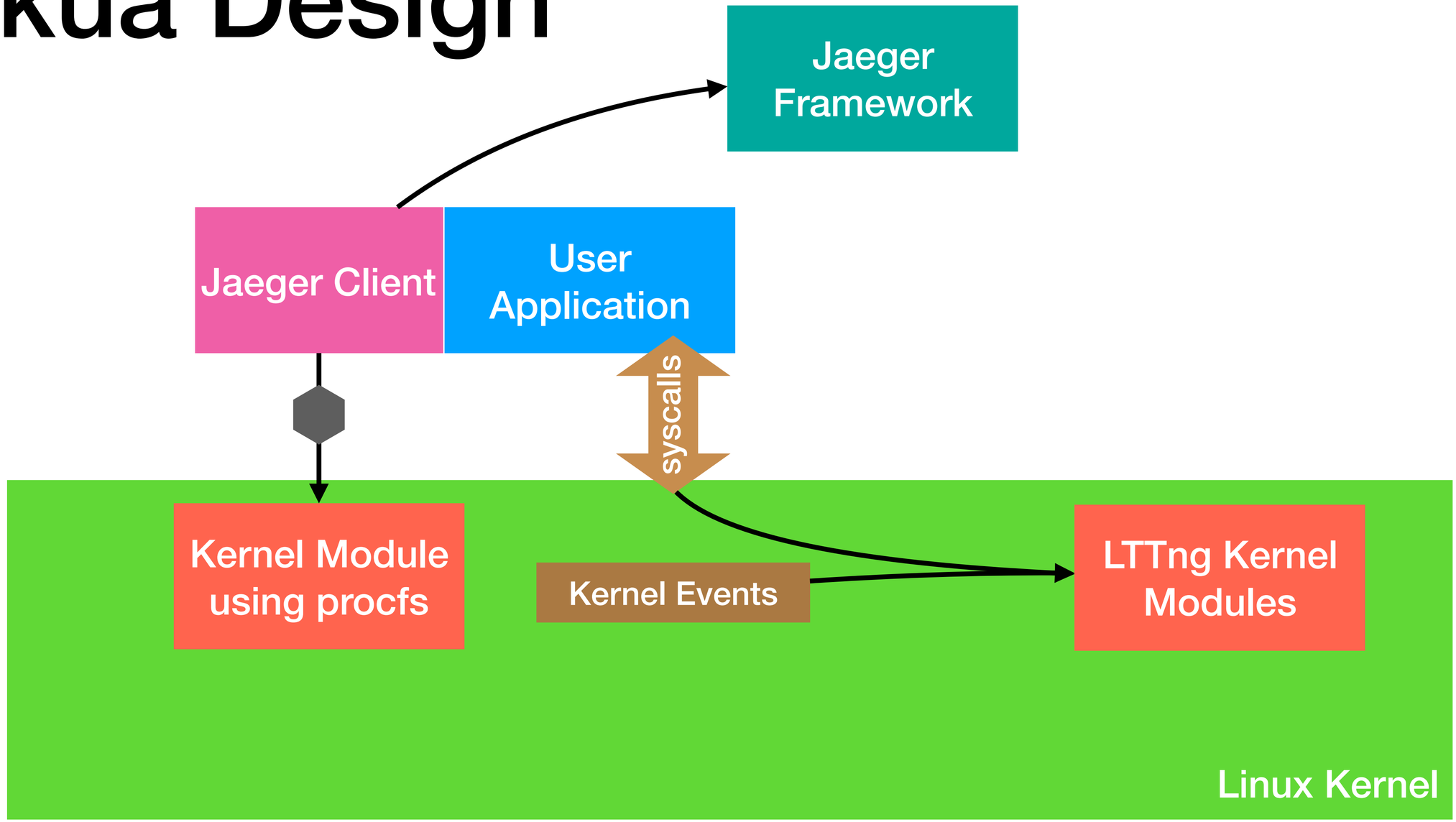
Skua Design


Jaeger
Framework
Jaeger Client
User
Application
syscalls
Kernel Events
Linux Kernel

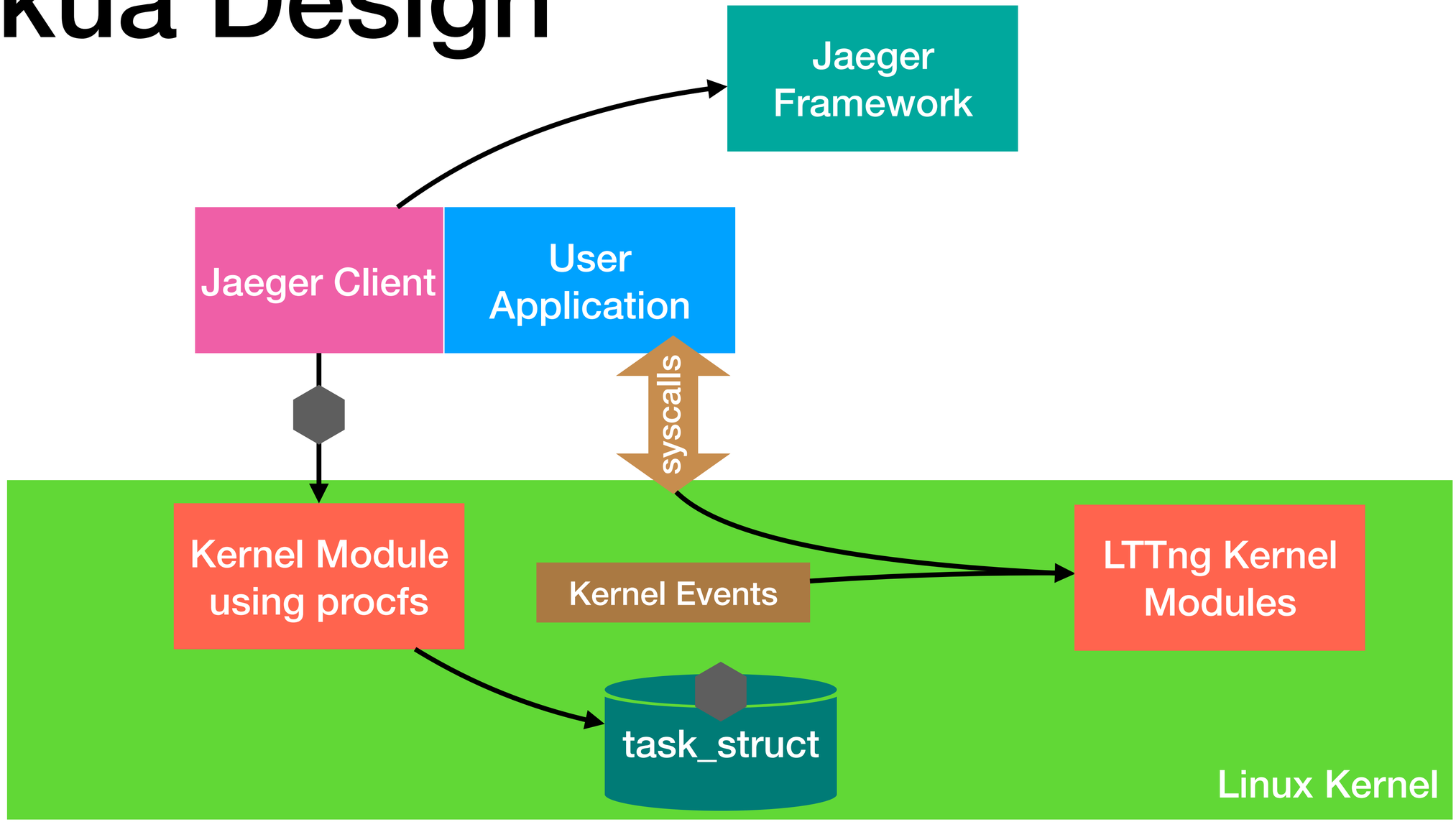
Skua Design


Jaeger
Framework
Jaeger Client
User
Application
syscalls
Kernel Events
LTTng Kernel
Modules
Linux Kernel

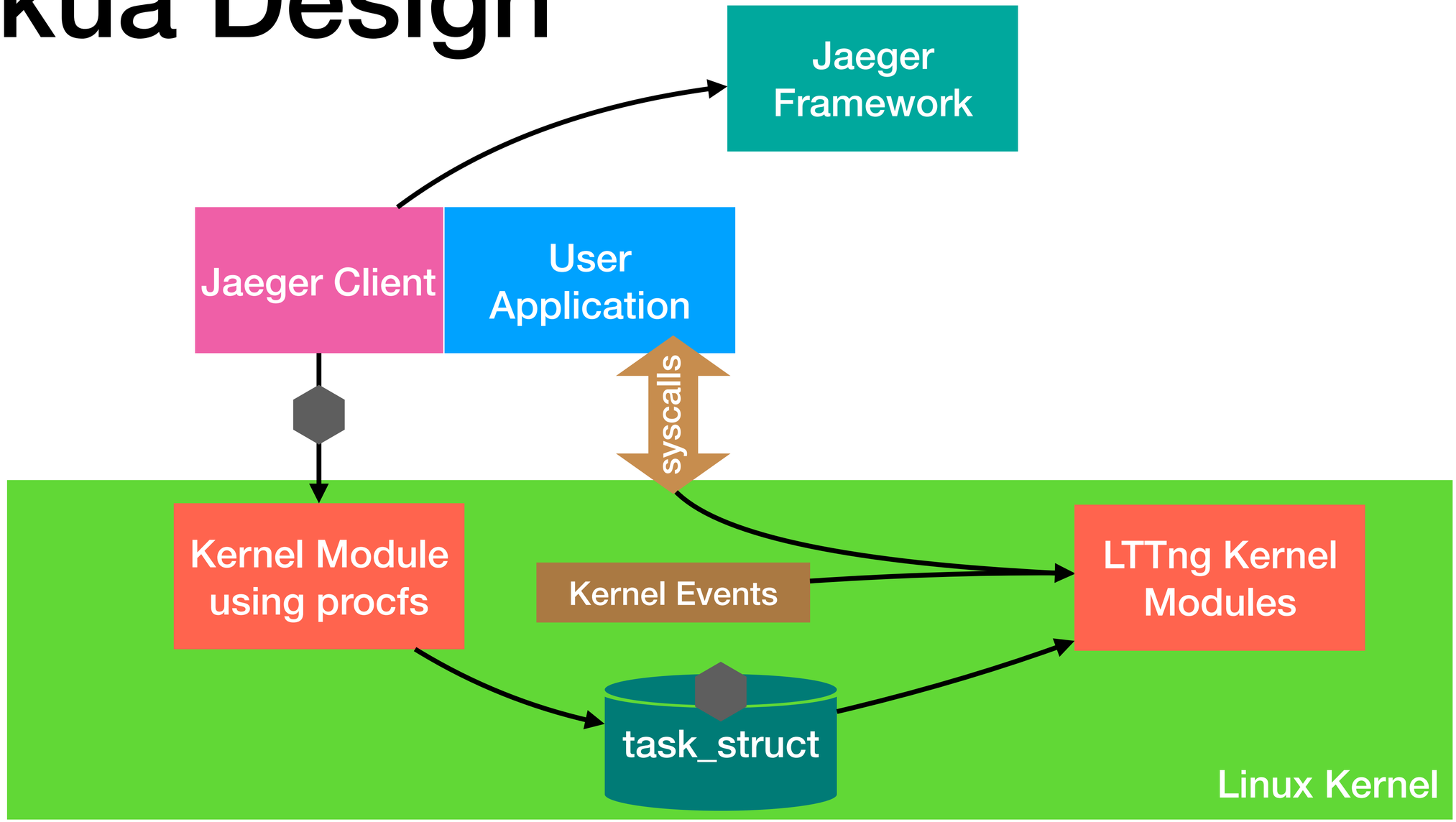
Skua Design


Jaeger
Framework
Jaeger Client
User
Application
syscalls
Kernel Module
using procfs
Kernel Events
LTTng Kernel
Modules
Linux Kernel

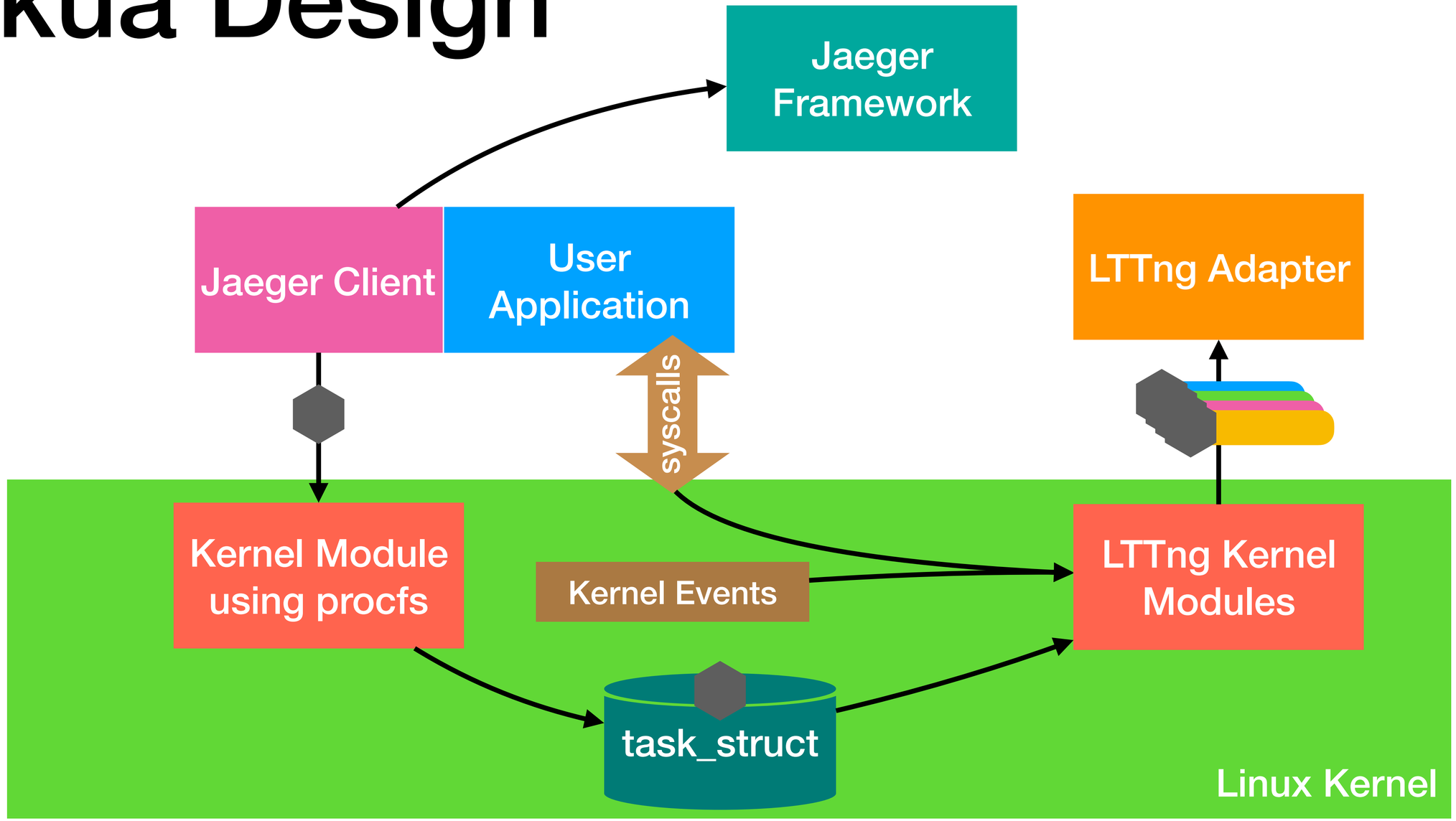
Skua Design


Jaeger
Framework
Jaeger Client
User
Application
syscalls
Kernel Module
using procfs
Kernel Events
LTTng Kernel
Modules
task_struct
Linux Kernel

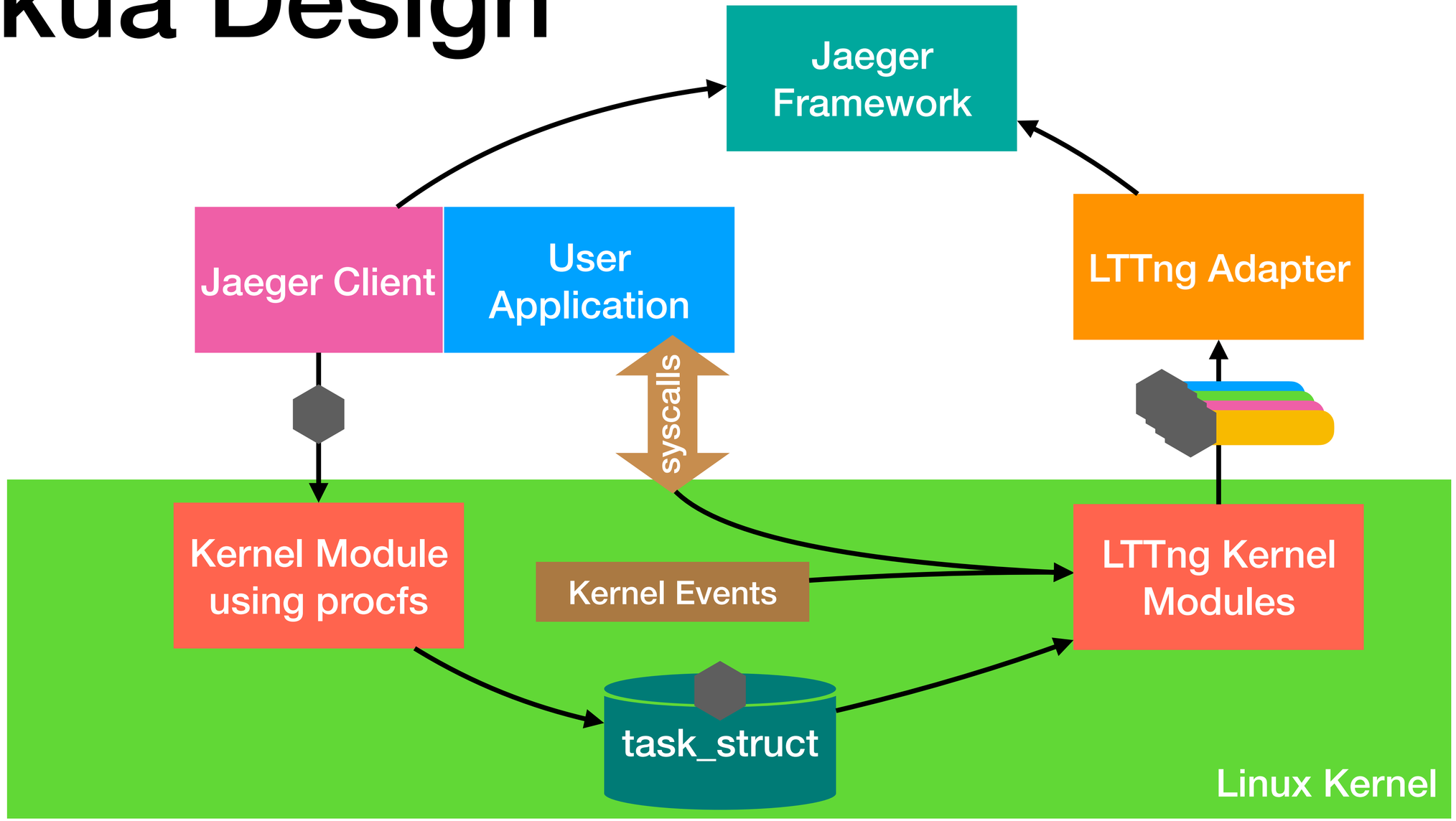
Skua Design


Jaeger
Framework
Jaeger Client
User
Application
syscalls
Kernel Module
using procfs
Kernel Events
LTTng Kernel
Modules
task_struct
Linux Kernel

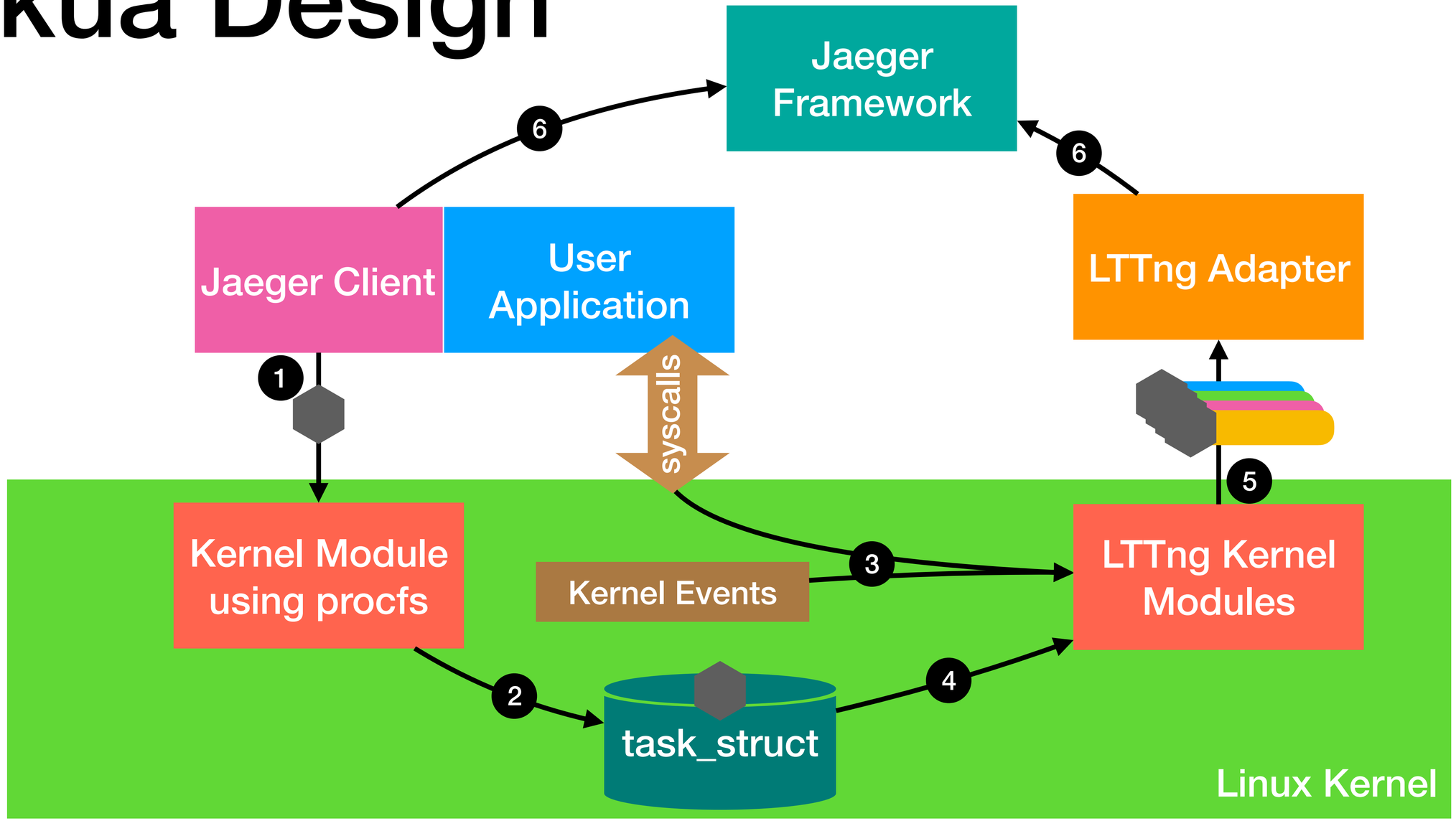
Skua Design


Jaeger Framework
Jaeger Client
User Application
LTTng Adapter
syscalls
Kernel Module using procfs
Kernel Events
LTTng Kernel Modules
task_struct
Linux Kernel

Skua Design


Jaeger Framework
Jaeger Client
User Application
LTTng Adapter
syscalls
Kernel Module using procfs
Kernel Events
LTTng Kernel Modules
task_struct
Linux Kernel

Skua Design


Jaeger Framework
6
6
Jaeger Client
User Application
LTTng Adapter
1
syscalls
5
Kernel Module using procfs
Kernel Events
3
LTTng Kernel Modules
2
4
task_struct
Linux Kernel

Skua Details

- Jaeger C++ client sends its context into the kernel 25 LOC
- Treats the Linux kernel as the next level of the span hierarchy
  - Each syscall is considered a span
  - Tracepoint events become span logs
- LTTng kernel modules tag each span and log with the context information 80 LOC
- Custom adapter sends kernel data into the Jaeger 250 LOC

Evaluation

Correctness

Setup

- Small C++ program
  - Spawns a few threads
  - Makes 10 different syscalls
- Verifies that Skua is correctly recording syscalls

Results

- Syscalls recorded in Jaeger as spans
- Misses a few syscalls
  - vDSO — `gettimeofday`
  - LTTng instrumentation
- Tracepoint events recorded properly as logs

Performance Benchmarks

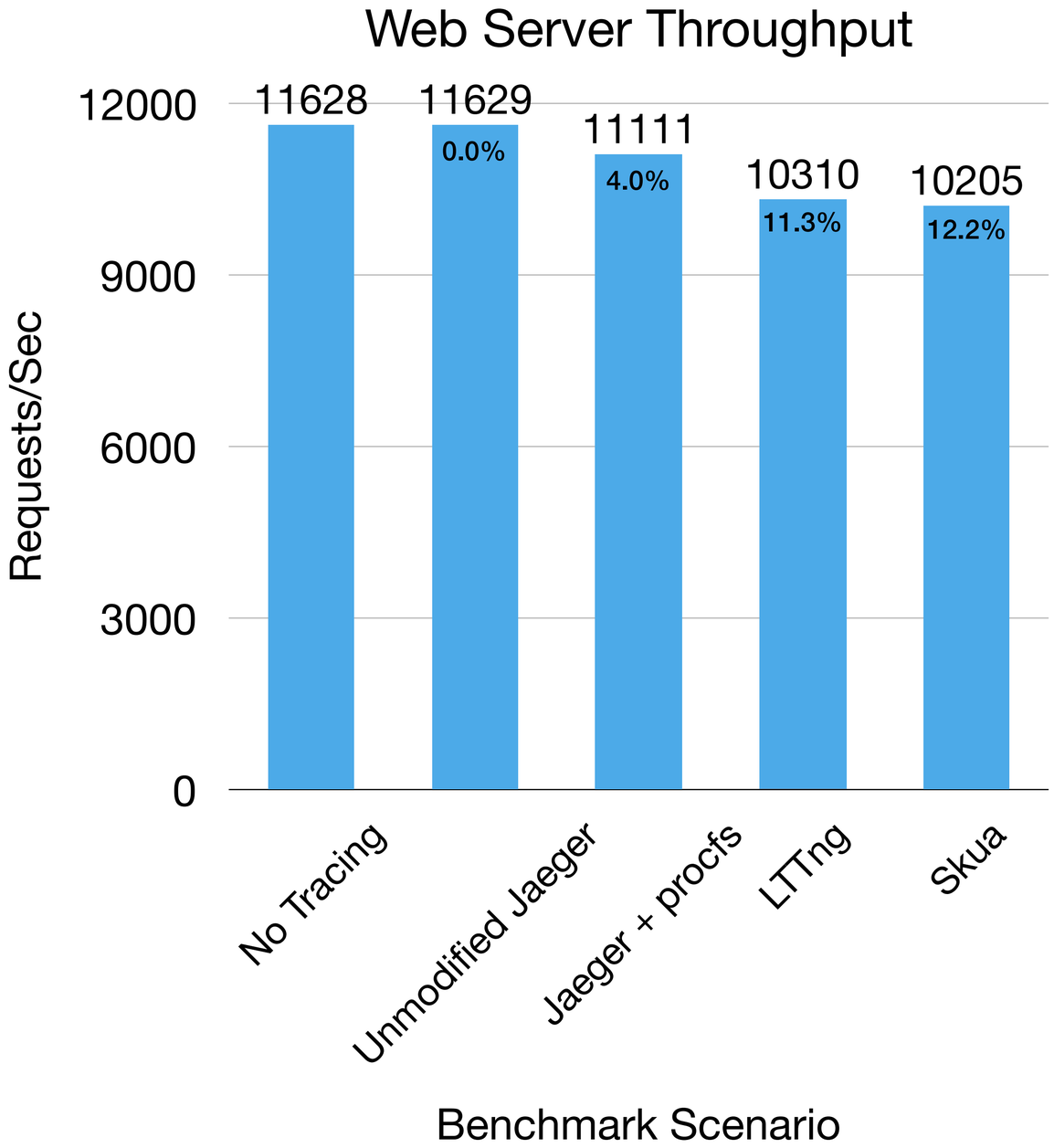
- Tests run on KVM virtual machine assigned 24 vCPUs (2 × Intel Xeon X5670), 16 GB RAM, Linux kernel 4.15.14 with Skua modifications

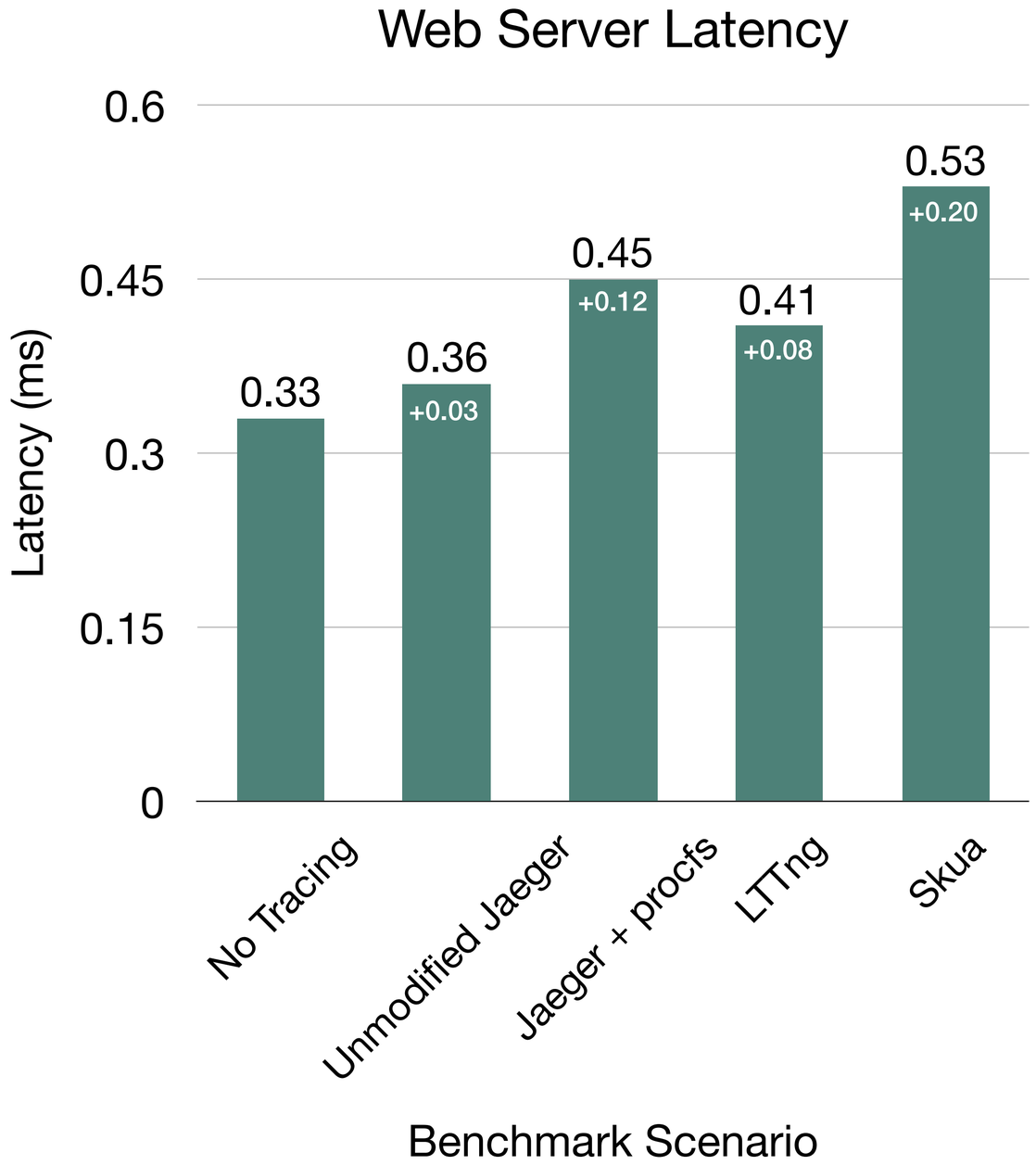
- Traced 0.1% of requests

Benchmark Scenario	Program Instrumentation	Kernel Tracing via Modified LTTng
No Tracing	None	No
Unmodified Jaeger	Original Jaeger client	No
Jaeger + procfs	Jaeger client modified to send trace context into kernel	No
LTTng	None	Yes, but output filtered using LTTng filters
Skua	Jaeger client modified to send trace context into kernel	Yes, output sent to adapter

Tiny HTTP Server

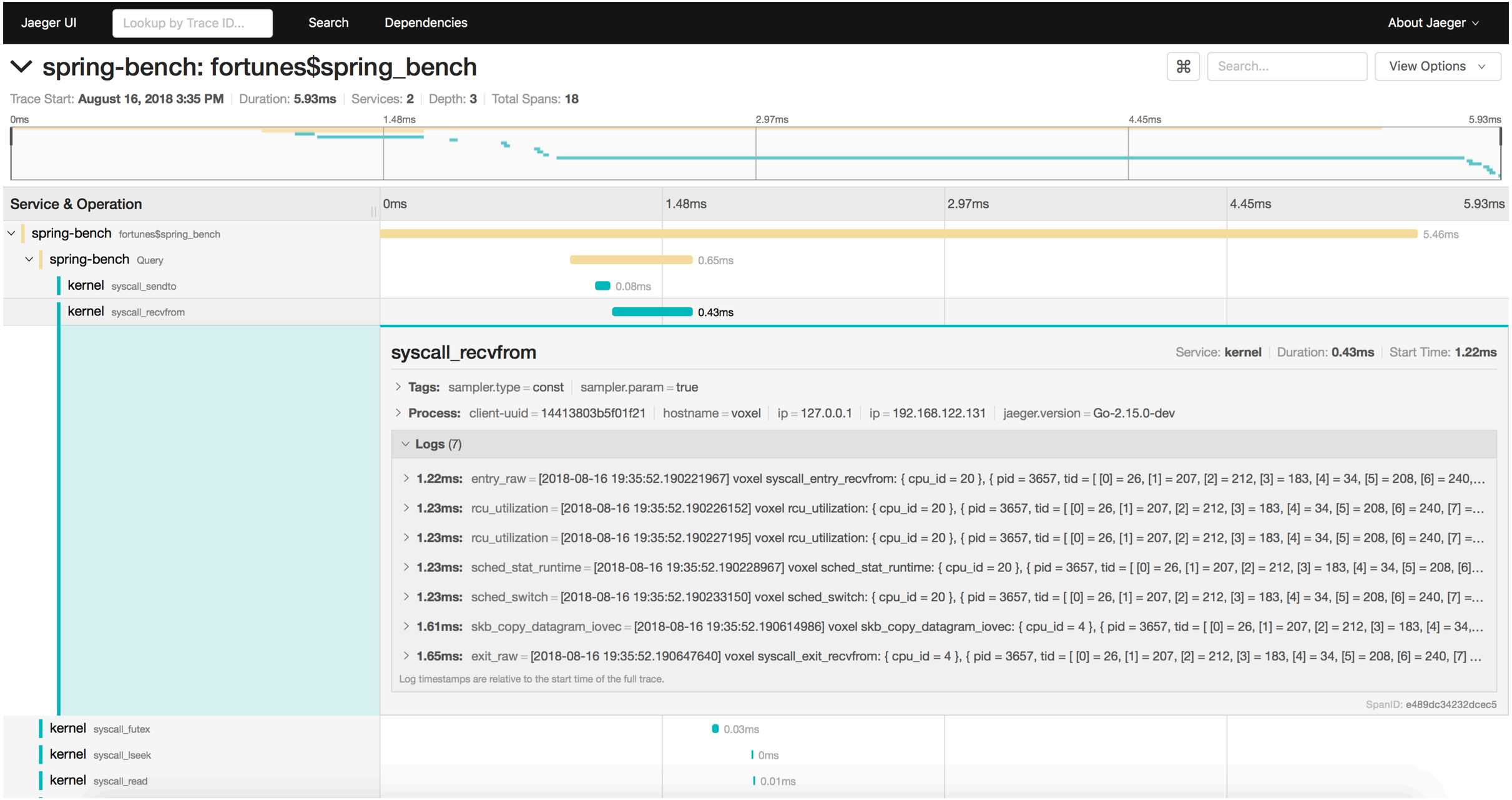
- Created a small C++ Web server using uWebSockets
- Used benchmarking tool autocannon
  - Sent 1 million GET requests using 10 connections
- Evaluated throughput and latency under different tracing scenarios

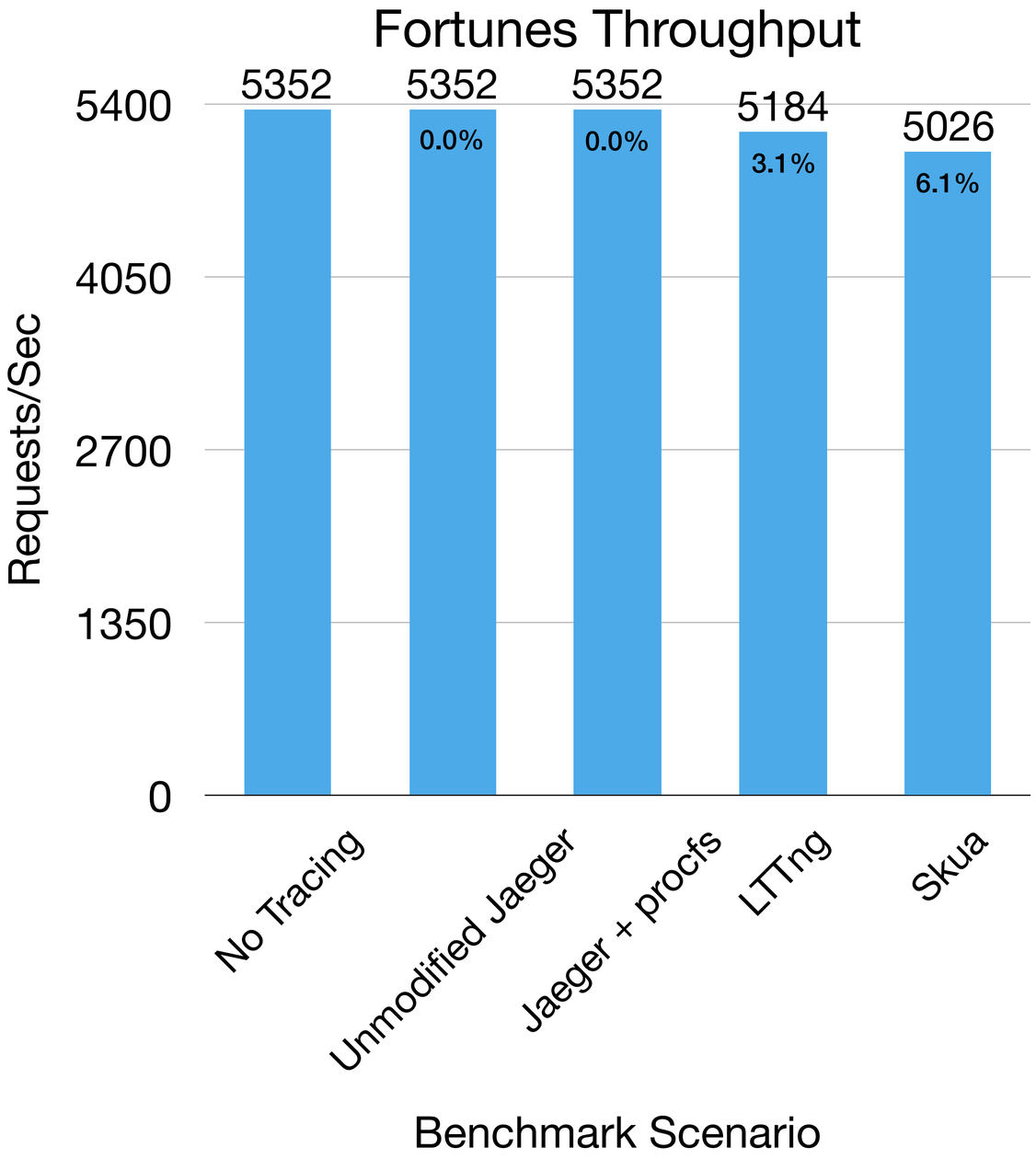

Web Server Throughput
Requests/Sec
12000
9000
6000
3000
0
11628
11629
0.0%
11111
4.0%
10310
11.3%
10205
12.2%
No Tracing
Unmodified Jaeger
Jaeger + procfs
LTTng
Skua
Benchmark Scenario

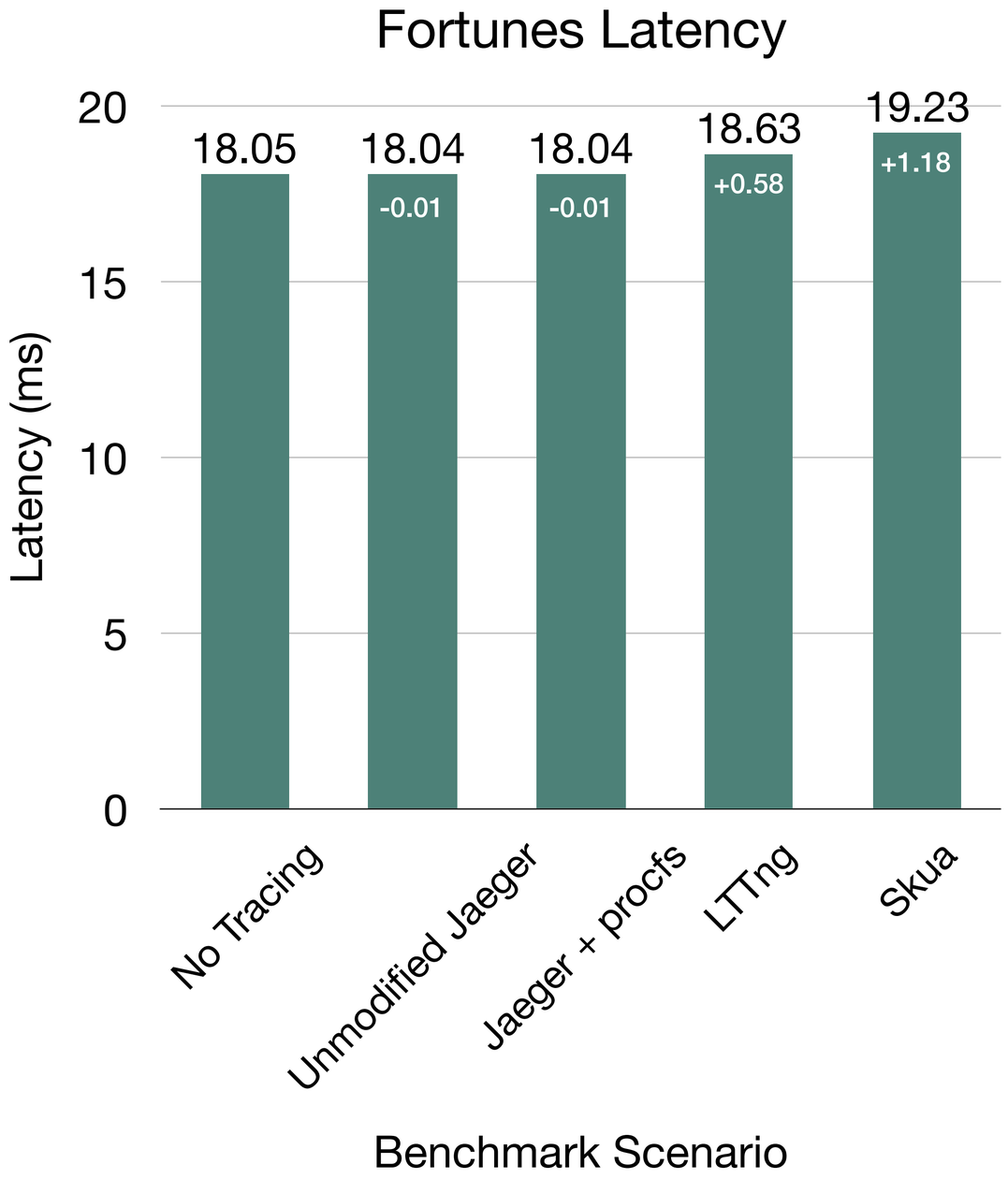

Web Server Latency
Latency (ms)
0.6
0.45
0.3
0.15
0
0.33
0.36
+0.03
0.45
+0.12
0.41
+0.08
0.53
+0.20
No Tracing
Unmodified Jaeger
Jaeger + procfs
LTTng
Skua
Benchmark Scenario

Fortunes Benchmark

- Code ~~stolen~~ borrowed from TechEmpower's Web Frameworks benchmark
  - Retrieves list of fortunes from database and renders HTML page
  - Uses Spring Boot, Kotlin, OpenJDK 10, PostgreSQL 10.4, OpenTracing integration
  - "Real-world" Web application
- Similar benchmarking process, but autocannon run twice for JIT warmup and with 100 connections


Jaeger UI
Lookup by Trace ID...
Search
Dependencies
About Jaeger
spring-bench: fortunes$spring_bench
Search...
View Options
Trace Start: August 16, 2018 3:35 PM | Duration: 5.93ms | Services: 2 | Depth: 3 | Total Spans: 18
0ms
1.48ms
2.97ms
4.45ms
5.93ms
Service & Operation
spring-bench fortunes$spring_bench
5.46ms
spring-bench Query
0.65ms
kernel syscall_sendto
0.08ms
kernel syscall_recvfrom
0.43ms
syscall_recvfrom
Service: kernel | Duration: 0.43ms | Start Time: 1.22ms
Tags: sampler.type = const | sampler.param = true
Process: client-uuid = 14413803b5f01f21 | hostname = voxel | ip = 127.0.0.1 | ip = 192.168.122.131 | jaeger.version = Go-2.15.0-dev
Logs (7)
1.22ms: entry_raw = [2018-08-16 19:35:52.190221967] voxel syscall_entry_recvfrom: { cpu_id = 20 }, { pid = 3657, tid = [[0] = 26, [1] = 207, [2] = 212, [3] = 183, [4] = 34, [5] = 208, [6] = 240,...
1.23ms: rcu_utilization = [2018-08-16 19:35:52.190226152] voxel rcu_utilization: { cpu_id = 20 }, { pid = 3657, tid = [[0] = 26, [1] = 207, [2] = 212, [3] = 183, [4] = 34, [5] = 208, [6] = 240, [7] =...
1.23ms: rcu_utilization = [2018-08-16 19:35:52.190227195] voxel rcu_utilization: { cpu_id = 20 }, { pid = 3657, tid = [[0] = 26, [1] = 207, [2] = 212, [3] = 183, [4] = 34, [5] = 208, [6] = 240, [7] =...
1.23ms: sched_stat_runtime = [2018-08-16 19:35:52.190228967] voxel sched_stat_runtime: { cpu_id = 20 }, { pid = 3657, tid = [[0] = 26, [1] = 207, [2] = 212, [3] = 183, [4] = 34, [5] = 208, [6]...
1.23ms: sched_switch = [2018-08-16 19:35:52.190233150] voxel sched_switch: { cpu_id = 20 }, { pid = 3657, tid = [[0] = 26, [1] = 207, [2] = 212, [3] = 183, [4] = 34, [5] = 208, [6] = 240, [7] =...
1.61ms: skb_copy_datagram_iovec = [2018-08-16 19:35:52.190614986] voxel skb_copy_datagram_iovec: { cpu_id = 4 }, { pid = 3657, tid = [[0] = 26, [1] = 207, [2] = 212, [3] = 183, [4] = 34,...
1.65ms: exit_raw = [2018-08-16 19:35:52.190647640] voxel syscall_exit_recvfrom: { cpu_id = 4 }, { pid = 3657, tid = [[0] = 26, [1] = 207, [2] = 212, [3] = 183, [4] = 34, [5] = 208, [6] = 240, [7] ...
Log timestamps are relative to the start time of the full trace.
SpanID: e489dc34232dcec5
kernel syscall_futex
0.03ms
kernel syscall_lseek
0ms
kernel syscall_read
0.01ms


Fortunes Throughput
Requests/Sec
5400
4050
2700
1350
0
5352
5352
0.0%
5352
0.0%
5184
3.1%
5026
6.1%
No Tracing
Unmodified Jaeger
Jaeger + procfs
LTTng
Skua
Benchmark Scenario


Fortunes Latency
Latency (ms)
20
15
10
5
0
18.05
18.04
-0.01
18.04
-0.01
18.63
+0.58
19.23
+1.18
No Tracing
Unmodified Jaeger
Jaeger + procfs
LTTng
Skua
Benchmark Scenario

Performance Overheads

- Unmodified Jaeger has a negligible impact on performance
- LTTng causes a moderate decrease in throughput and a small increase in latency
  - This could be improved by enabling a subset of available instrumentation points
- Our modifications to Jaeger cause additional latency (depending on scenario)
  - Performing syscalls to propagate the trace context is expensive
  - Ingestion of kernel events is more work
- In the tiny HTTP benchmark, Web server transactions took under 1ms each, causing the latency impacts to appear large by comparison

Future Work

- Improve performance
- Simplify installation process
- Adaptive sampling reconfiguration
- Attempt tracing Ceph with Skua

Logging What Matters: Just-In-Time Instrumentation And Tracing
Lily Sturmann, Emre Ates
Friday, Aug. 17 4:30pm

Tracing Ceph using Jaeger-Blkkin
Mania Abdi
Saturday, Aug. 18 12:00pm

Conclusions

- Can use distributed tracing to monitor and debug complex distributed systems
- Current open source distributed tracing frameworks miss kernel information
- Skua integrates LTTng kernel data with Jaeger tracing
- Skua has some impact on throughput and latency

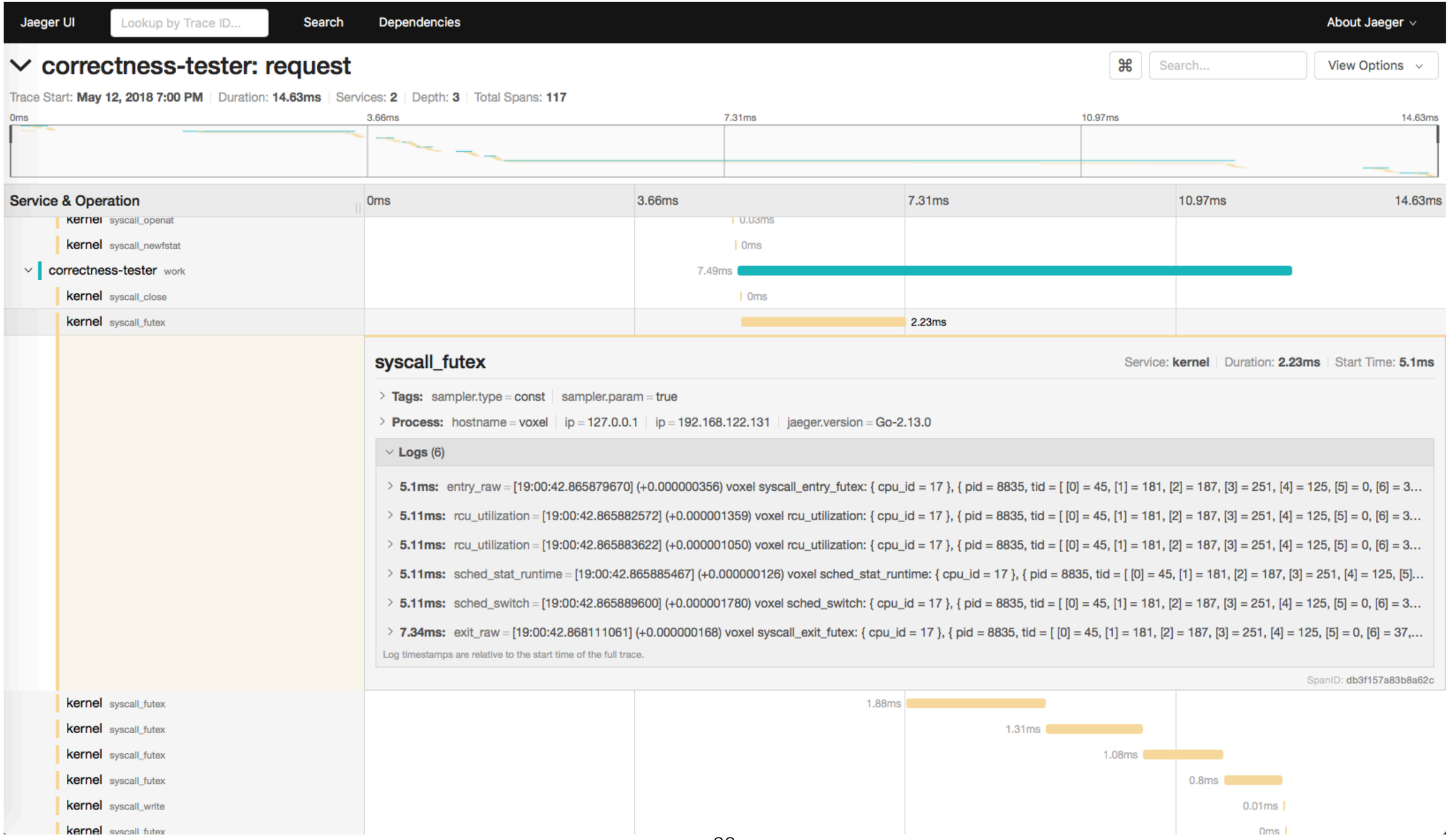
Acknowledgements

- Raja Sambasivan - Mentor
- 

Backup Slides

Correctness Tester Syscalls

- `getpid`
- `getppid`
- `gettid`
- `gettimeofday`
- `nanosleep`
- `open`
- `close`
- `write`
- `fstat`
- `futex`


Jaeger UI
Lookup by Trace ID...
Search
Dependencies
About Jaeger
correctness-tester: request
Search...
View Options
Trace Start: May 12, 2018 7:00 PM | Duration: 14.63ms | Services: 2 | Depth: 3 | Total Spans: 117
0ms
3.66ms
7.31ms
10.97ms
14.63ms
Service & Operation
kernel syscall_openat
kernel syscall_newfstat
correctness-tester work
kernel syscall_close
kernel syscall_futex
0.03ms
0ms
7.49ms
0ms
2.23ms
syscall_futex
Service: kernel | Duration: 2.23ms | Start Time: 5.1ms
Tags: sampler.type = const | sampler.param = true
Process: hostname = voxel | ip = 127.0.0.1 | ip = 192.168.122.131 | jaeger.version = Go-2.13.0
Logs (6)
5.1ms: entry_raw = [19:00:42.865879670] (+0.000000356) voxel syscall_entry_futex: { cpu_id = 17 }, { pid = 8835, tid = [[0] = 45, [1] = 181, [2] = 187, [3] = 251, [4] = 125, [5] = 0, [6] = 3...
5.11ms: rcu_utilization = [19:00:42.865882572] (+0.000001359) voxel rcu_utilization: { cpu_id = 17 }, { pid = 8835, tid = [[0] = 45, [1] = 181, [2] = 187, [3] = 251, [4] = 125, [5] = 0, [6] = 3...
5.11ms: rcu_utilization = [19:00:42.865883622] (+0.000001050) voxel rcu_utilization: { cpu_id = 17 }, { pid = 8835, tid = [[0] = 45, [1] = 181, [2] = 187, [3] = 251, [4] = 125, [5] = 0, [6] = 3...
5.11ms: sched_stat_runtime = [19:00:42.865885467] (+0.000000126) voxel sched_stat_runtime: { cpu_id = 17 }, { pid = 8835, tid = [[0] = 45, [1] = 181, [2] = 187, [3] = 251, [4] = 125, [5]...
5.11ms: sched_switch = [19:00:42.865889600] (+0.000001780) voxel sched_switch: { cpu_id = 17 }, { pid = 8835, tid = [[0] = 45, [1] = 181, [2] = 187, [3] = 251, [4] = 125, [5] = 0, [6] = 3...
7.34ms: exit_raw = [19:00:42.868111061] (+0.000000168) voxel syscall_exit_futex: { cpu_id = 17 }, { pid = 8835, tid = [[0] = 45, [1] = 181, [2] = 187, [3] = 251, [4] = 125, [5] = 0, [6] = 37,...
Log timestamps are relative to the start time of the full trace.
SpanID: db3f157a83b8a62c
kernel syscall_futex
kernel syscall_futex
kernel syscall_futex
kernel syscall_futex
kernel syscall_write
kernel syscall_futex
1.88ms
1.31ms
1.08ms
0.8ms
0.01ms
0ms

Existing Tracing Frameworks

- Dapper (2010) — Google
- Zipkin (2012) —Twitter
- Canopy (2017) — Facebook
- Jaeger (2017) — open sourced by Uber